A Modular Approach to Object Initialization for Pharo

Marco Naddeo

UNIVERSITÀ DEGLI STUDI DI TORINO

di.unito.it

DIPARTIMENTO DI INFORMATICA

Doctoral schools

Scuola di Dottorato in Scienze della Natura e Tecnologie Innovative – Università degli Studi di Torino

École Doctorale Sciences Pour l'Ingénieur – Université Lille Nord-de-France

---

Marco Naddeo

A Modular Approach to Object Initialization for Pharo

PhD thesis completed in "co-tutelle"

Composition of the jury

Viviana Bono	Professor at the University of Torino	Italy	Supervisor
Stéphane Ducasse	Director of Research at INRIA Lille	France	Supervisor
Oscar Nierstrasz	Professor at the University of Bern	Switzerland	Reviewer
Julien Ponge	Professor at INSA Lyon Telecom	France	Secretary
Aleksy Schubert	Professor at the University of Warsaw	Poland	Reviewer
Elena Zucca	Professor at the University of Genova	Italy	Chair

---

24 November 2017

To my beloved Dad

Abstract

English

Code modularity is important for code reuse. Language designers mainly focused on method code modularity. On the contrary, initialization code modularity has usually been neglected. The initialization code of many class-based object-oriented languages becomes cumbersome in different situations: for example, when some fields have *multiple initialization options* (*e.g.,* a rectangle can be positioned by providing Cartesian or polar coordinates), have *optional initialization* (*e.g.,* a rectangle can be positioned in the origin `(0,0)` if no other coordinates are provided), or have *superclass initialization* (*e.g.,* a colored rectangle must redefine all the constructors of its rectangle superclass to add the notion of color). In such cases, the number of constructors increases exponentially in the size of the initialization options, since the initialization approach lacks modularity.

We propose a novel approach to field initialization, inspired by the prototypical language Magda. This approach is based on easy-to-compose initialization modules, which result to be smaller compared to classical constructors. Moreover, their number grows linearly in the size of the initialization options. We apply such approach to Pharo, a dynamically-typed object-oriented programming language inspired by Smalltalk. The adaptation to Pharo of Magda's initialization approach implies solving some new challenges related to moving from a statically typed context to a dynamically typed one.

Français

La modularité du code favorise la réutilisation. Les concepteurs de langages de programmation se sont principalement consacrés à la modularité des méthodes. À l'inverse, la modularité du code d'initialisation a souvent été négligé. Le code d'initialisation de beaucoup de langages orientés objets à classe est laborieux dans plusieurs cas: par exemple, quand quelques champs (*i.e.,* variables d'instances) ont *plusieurs options d'initialisation* (*e.g.,* un rectangle peut être initialisé avec des coordonnées Cartésiennes ou polaires), ont une *initialisation optionnelle* (*e.g.,* un rectangle peut être positionné à une coordonnée fournie ou à l'origine `(0,0)` si aucune coordonnée n'est fournie) ou héritent d'une *initialisation de la super classe* (*e.g.,* un rectangle coloré doit redéfinir tous les constructeurs de sa super classe pour ajouter la notion de couleur). Dans de telles circonstances, le nombre de constructeurs augmente exponentiellement avec les options d'initialisation car l'approche d'initialisation manque de modularité.

Nous proposons une approche d'initialisation novatrice s'inspirant du langage prototypique Magda. Cette approche est basée sur des modules d'initialisation composables plus petits que les constructeurs classiques. De plus, le nombre de ces modules augmente linéairement avec les options d'initialisation. Nous appliquons cette approche à Pharo, un langage orienté objets dynamiquement typé inspiré de Smalltalk. L'adaptation à Pharo de l'approche d'initialisation de Magda implique la résolution de nouveaux défis issus du passage d'un contexte statiquement typé à un contexte dynamique.

Italiano

La modularità del codice è importante per il riuso del codice. I progettisti di linguaggi si sono focalizzati principalmente sulla modularità del codice dei metodi, ma, al contrario, la modularità del codice di inizializzazione è stata di solito trascurata. Il codice di inizializzazione di molti linguaggi object-oriented basati su classi diventa ingombrante in diverse situazioni: ad esempio, quando alcuni campi hanno *opzioni di inizializzazione multiple* (ad esempio, un rettangolo può essere posizionato fornendo delle coordinate cartesiane o polari), hanno una *inizializzazione opzionale* (ad esempio, un rettangolo può essere posizionato nell'origine `(0,0)` se non vengono fornite altre coordinate), o ereditano una *inizializzazione della superclasse* (ad esempio, un rettangolo colorato deve ridefinire tutti i costruttori della sua superclasse rettangolo per aggiungere la proprietà colore). In questi casi, il numero di costruttori aumenta esponenzialmente nel numero delle opzioni di inizializzazione, siccome l'approccio di inizializzazione manca di modularità.

Proponiamo un nuovo approccio all'inizializzazione dei campi, ispirato dal linguaggio prototipale Magda. Tale approccio è basato su moduli di inizializzazione facilmente componibili, che risultano essere più piccoli in confronto ai costruttori tradizionali. Inoltre, il loro numero cresce linearmente con il numero di opzioni di inizializzazione. Applichiamo questo approccio a Pharo, un linguaggio di programmazione object-oriented dinamicamente tipato ispirato da Smalltalk. L'adattamento a Pharo dell'approccio di inizializzazione di Magda implica risolvere alcune nuove sfide relative all'andare da un contesto staticamente tipato verso uno dinamicamente tipato.

Contents

Introduction **1**

1 Limits of Pharo's initialization approach **4**
1.1 Initializing objects in Pharo and related problems 4
1.1.1 The `initialize` method 4
1.1.2 Instance-creation methods 5
1.1.3 Lazy initialization 7
1.2 The number of constructors explosion 7
1.2.1 Multiple initialization options 7
1.2.2 Optional initialization 8
1.2.3 Problems with subclassing 10

2 The ini-module model **12**
2.1 What is an ini-module and some definitions 12
2.2 A semantics for ini-module activation 16
2.3 The algorithm 29
2.4 Note on `I2` instructions 36
2.5 Discussion on the model 36
2.5.1 Design choices 37
2.5.2 Checks on well-formedness of ini-modules 38
2.5.3 Class invariants 40

3 The ordering algorithm **43**
3.1 Directly specifying a total ordering 43
3.2 Specifying a partial ordering and inferring a linear extension 45
3.2.1 Formal definition of the partial order $\leq$ 49
3.2.2 Calculating a linear extension of the partial order $\leq$ 53

4 Pharo implementation **60**
4.1 Ini-modules in Pharo 60
4.1.1 Extending the initialization protocol in subclasses 62
4.1.2 How to specify explicit constraints 63
4.1.3 Graphical user interface for ini-modules 66

4.2 Ini-modules as methods . . . 67
4.2.1 Drawbacks of the discussed approach . . . 68

5 Evaluation **71**
5.1 Solving the initialization problems of Pharo . . . 71
5.2 Solving the number of constructors explosion . . . 72
5.2.1 Multiple initialization options . . . 72
5.2.2 Optional initialization . . . 72
5.2.3 Problems with subclassing . . . 73

6 Related work **75**
6.1 Common Lisp / CLOS . . . 75
6.2 Constructors and methods with named and optional parameters . . . 75
6.3 Other approaches to initialization . . . 76
6.4 Other works related to initialization . . . 79

7 Future work **81**
7.1 Extending our analysis to other dynamic languages . . . 81
7.2 Maximising ini-modules reuse . . . 82
7.3 Performance . . . 82
7.4 Going beyond the limitations of the prototype . . . 83

Acknowledgements **86**

Introduction

Code reuse is a vital issue and it is one of the main promises of object-orientation, therefore object-oriented programming languages must strive to make it as easy as possible. Language designers have mainly focused on reuse of object's behavior, *i.e.,* methods. Pharo [3], a source dialect of Smalltalk and implementation of its programming environment, is an example of a language whose community put a lot of effort in the modularization of code, for instance, by introducing *traits* [9].

However, language designers have usually ignored the reuse of field-initialization protocols, *i.e.,* the entirety of conventional rules which establish how an object is initialized at creation time. In this sense, no emphasis has been put by the Pharo community on the modularization of initialization protocols yet. Indeed, Pharo has a poor initialization protocol close to those of mainstream object-oriented class-based statically-typed languages, as Java.

In fact, Pharo only equips the class designer with an `initialize` method, which is useful to initialize fields to default values. However, if the class user must be able to specify values at object creation, then the developer is forced to implement dedicated methods on class side (*i.e., static methods* in Java parlance). This workaround provides for the lacking of Pharo's initialization protocol but has various problems.

Other problems are more generally concerned to the non-modularity of the initialization protocol. Indeed, traditional initialization protocols require the class designer to specify in a unique block of code how an object can be initialized. If a class has n sets of fields that can be initialized in m different ways, then the class designer is forced to manually implement all the combinations. This results in an exponential explosion of the number of constructors. Moreover, if a class has one or more fields that may or may not be initialized at construction time, the number of constructors should be multiplied by two for each optionally-initialized field, if the language does not support optional initialization. Finally, extending the initialization protocol of a class while designing a subclass with new fields may result in an annoying task, when the initialization protocol of the superclass is not inherited. All these problems are common both to Java and Pharo, for example.

Kuśmierek proposed a solution to these problems in Magda [13, 4], a statically-typed prototypical language. Magda features a *modular initialization protocol* approach based on small, composable units called *initialization modules* (*ini-modules* from now on). Each ini-module may contain the initialization code for one or more fields, and it has a set of formal parameters, called *input parameters*, which can be supplied at object creation. An ini-module also may specify a set of *output parameters*, referring by name to input parameters declared

in other modules, which will be computed by the ini-module and supplied to such other ini-modules. Intuitively, the ini-module composition mechanism is equivalent to implementing all valid initialization protocols, which can be a tedious task when done manually. The composition mechanism acts like a Cartesian product of the desired initialization options for the fields. Magda ini-modules are a general enough construct (for instance, they were applied to Java [6]), and look promising to be introduced in dynamically-typed programming languages.

In this thesis, we tailor Magda ini-modules to the Pharo language. The adaptation of ini-modules to Pharo implies solving some new challenges, essentially related to the *ease of use*. Indeed, Magda ini-modules are equipped with static checks that impose constraints on how Magda code is written: this is something we want to avoid in the Pharo setting as much as possible, in order to keep its inherent ease of use. We describe the Pharo ini-module model in the hope it will be useful to other dynamic-language designers.

Contributions of the thesis

The main contributions of this thesis are:

- a study of the initialization problems of Pharo;
- an adaptation of Kuśmierek's approach of ini-modules to the dynamic nature of Pharo;
- an algorithm to derive a linear ordering of ini-module activation, based on topological sorting;
- a prototype of Pharo extended with ini-modules, available at http://smalltalkhub.com/#!/~MarcoNaddeo/IniModules.

Outline of the thesis

The thesis is organized as follows:

- a detailed description of the object initialization techniques used in Pharo and their problems, along with a brief description of the problems inherent to the standard object initialization techniques; starting from these problems, we extract a set of requirements for our solution (Chapter 1);
- a description of the ini-module model as adapted for Pharo and the differences with the original model of Magda (Chapter 2);
- a discussion about how the class designer can impose the ordering in which the ini-modules defined in a class must be considered for execution (Chapter 3);
- a presentation of our Pharo implementation (Chapter 4);

- an evaluation of our solution with respect to the previously identified requirements (Chapter 5);
- a summary of the related work concerning both statically and dynamically-typed languages (Chapter 6);
- a discussion on future work (Chapter 7).

Chapter 1

Limits of Pharo's initialization approach

Section 1.1 presents Pharo's initialization protocol and its limitations. We also use this section as an introduction to the Pharo language. Then, Section 1.2 extends the analysis to more general problems, which are well-known and have already been pointed out for statically-typed languages [13, 4, 15]. We introduce them here by focusing on Pharo.

While discussing the main problems with the traditional initialization approach, we also extract some requirements for our solution, that we will show to be met in Chapter 5.

1.1 Initializing objects in Pharo and related problems

1.1.1 The `initialize` method

Pharo classes may override the `initialize` method to specify how new instances are initialized. Let us consider a `TextArea` class in Pharo, whose instances are described by four fields:

- `rows`/`columns`, which both default to 0;
- `text`, which defaults to the empty string;
- `scrollbars`, which defaults to `ScrollbarsBoth`.

This method initializes fields to default values:

```
TextArea>>initialize
  super initialize.
  rows := 0.
  columns := 0.
  text := ''.
  scrollbars := Scrollbars both
```

This code shows the `initialize` method of the `TextArea` class. With this code, creating a new instance of the `TextArea` class with all fields initialized to default values can be done by sending the message `new` to the class:

```
textArea1 := TextArea new.
```

A limitation of the `initialize` method is that the class designer cannot specify any parameter to be supplied by the class user and is thus forced to initialize fields to default values.

Requirement 1 (Parameter passing)**.** We want our solution to allow the class designer to deal with parameters supplied at object-creation time, when this is needed.

1.1.2 Instance-creation methods

While the `initialize` solution is convenient, the developer creating a new text area could prefer to have some fields initialized to client-specified values. For this to work, dedicated class-side (*i.e., static methods* in Java parlance) *instance-creation methods* must be implemented in the `TextArea` metaclass:

```
TextArea class>>newWithText: aString
  "Create a new text area with a content of aString and default values for the rest"
  | instance |
  instance := self new.
  instance text: aString.
  ^ instance

TextArea class>>newWithText: aString
                rows: aNumber columns: anotherNumber
  "Create a new text area with a size specified by aNumber and anotherNumber, a content
   of aString, and default values for the rest"
  ^ self new
      rows: aNumber;
      columns: anotherNumber;
      text: aString;
      yourself
```

An instance-creation method (or *constructor method* [1]) is a method that creates a new object and sets some of the object's fields to user-specified values by sending *mutating messages* (*i.e.,* setters such as `rows:`, `columns:` and `text:`). It is important to note that in Pharo, contrary for example to Java, a static method of a class cannot access a field of an instance of this class. That is why an instance-creation method must go through mutating messages to initialize the instance it creates.

The `newWithText:` method (the first method of above code) in the `TextArea class` class (the metaclass of `TextArea`) returns (with the caret `^` symbol) the value of the `instance` temporary variable. This temporary variable is assigned the result of sending `new` to `self` (`self` is equivalent to `this` in Java): here `self` represents the `TextArea` class, as in Pharo classes are objects too. Because `self` represents the current class, sending `new` results in the creation of a new instance of that class. After sending the `new` message, the `text:` message is sent to set the text area's initial text to the one passed as the `aString` parameter.

The `newWithText:rows:columns` method (the second method of above code) creates a new `TextArea` instance with `rows`, `columns` and `text` fields initialized to user specified values. This method also shows some more Pharo syntax: instead of creating a temporary variable

and returning it as in `newWithText:`, the new object is directly returned. The semi-colon `';'` (named the *cascade* operator) is used to specify several messages to be sent to the same object (here the new instance). The `yourself` message just makes sure that the object returned by the constructor is the new instance and not the value that the `text:` method returns.

The class user now has two more ways to create text areas:

```
textArea2 := TextArea newWithText: 'some text'.
textArea3 := TextArea newWithText: 'some text' rows: 3 columns: 20.
```

Executing the above code will result in `textArea2` having default numbers of rows and columns and default scrollbars, whereas `textArea3` will only have default scrollbars.

This approach, however, presents some problems.

First of all, if the field values can be specified at construction time by the class user through instance-creation method parameters, the class designer must implement mutating methods.

Requirement 2 (Direct access to object fields)**.** We want our solution to be able to directly access object fields.

Moreover, implementing mutating methods increases the size of the public API of the class. In fact, in Pharo all methods are public. This is not satisfactory if the class designer would prefer to have unchanging values for some fields (*i.e., final fields* in Java parlance).

Requirement 3 (No pollution of the class' API)**.** We want our solution not to require implementing any supporting method that pollutes the class' API.

A workaround for this problem is to use the *Constructor Parameter Method* best practice pattern [1] that replaces all sent mutating messages by just one combining message: in our example, instead of adding the `rows:`, `columns:` and `text:` methods to the `TextArea` class' public API, the class designer could add only one `setRows:columns:text:` method. As a result, the API is less polluted: one public method instead of three. However, this workaround has to be re-applied for each instance-creation method.

Requirement 4 (Constructors activation context)**.** Constructors shall not be callable from outside the initialization process.

Another problem is that each instance-creation method starts by sending the `new` message to `self`. This automatically triggers a call to the `initialize` method setting fields to their default values. This happens even if some of these fields are immediately overwritten by sending a mutating message from an instance-creation method. For example, while executing the `newWithText:rows:columns:` method, the `initialize` method (shown in Section 1.1.1) sets four fields to default values of which three are immediately changed. This default initialization is useless and can be costly in terms of space and time.

Requirement 5 (Complete control over the initialization process)**.** We want our solution to leave the class designer complete control over the initialization process. This means that no initialization shall take place implicitly, or at least it shall always be possible for the class designer to turn it off.

A workaround for this problem is for the instance-creation method to send `basicNew` instead of `new` and thus avoid the call to `initialize`. Nevertheless, this workaround completely by-passes the initialization protocol of the whole hierarchy, not just the one of the class being instantiated. This means that fields with no client-specified values will not be initialized to default values.

1.1.3 Lazy initialization

A better solution to the just exposed problem of tight dependency between the `initialize` method and the instance-creation methods is to use the *Lazy Initialization* best practice pattern [1]. In this pattern, which represents a common practice among Pharo developers, the `initialize` method is restricted to its minimum and getter methods are responsible for initializing fields on demand. For example, we can write lazy-initializing getter methods for the `TextArea` class in this way:

```
TextArea>>rows
  ^ rows ifNil: [rows := 0]

TextArea>>columns
  ^ columns ifNil: [columns := 0]

TextArea>>text
  ^ text ifNil: [text := '']

TextArea>>scrollbars
  ^ scrollbars ifNil: [scrollbars := Scrollbars both]
```

Lazy initialization is typically accomplished by augmenting a field's getter method to check for a previously-setted value. If none exists, a default value is placed into the field, and this default value is returned to the caller in a just-in-time fashion.

In this manner object creation is deferred until first use which can, in some circumstances (*e.g.,* sporadic object access), increase system responsiveness and speed startup by bypassing large-scale object pre-allocation. Another advantage of lazy initialization is to completely avoid the initialization of a unused field. Nonetheless, this pattern solves no other discussed problem.

1.2 The number of constructors explosion

1.2.1 Multiple initialization options

A first problem with common, non-modular approaches to initialization is that they lead to an exponential number of constructors with respect to the number of class fields with different initialization options. Let us consider a `ColoredPoint` class with 2 sets of fields (5 in total):

- `x` and `y` can be initialized from Cartesian coordinates, polar coordinates or complex coordinates;

- `r`, `g` and `b` can be initialized from an RGB or CMYK tuple.

This situation induces, in a language like Pharo, six constructors, *i.e.,* instance-creation methods (see Figure 1.1). In more general terms, a class having n sets of fields, each with m_i initialization options, requires $\prod_{i=1}^{n} m_i$ instance-creation methods.

Requirement 6 (Multiple initialization options)**.** If a class has n sets of fields, each with m_i initialization options, we want our solution to allow the class designer to initialize them by implementing $O(\sum_{i=1}^{n} m_i)$ constructors.

In practice though, the class designer will just not implement some potentially valid constructors, letting the class user write more complex code than necessary. Moreover, as can be seen in Figure 1.1, this scheme typically introduces duplicated code of the field assignments.

In some cases, grouping sets of fields into their own classes (*e.g.,* a `Color` class here) will both solve the problem and lead to a better design. This is not always possible though.

1.2.2 Optional initialization

Many class-based object-oriented languages do not allow developers to explicitly manage optional parameters of methods: when some parameters are optional, the developer is forced to declare one constructor with all the parameters and one constructor for each legal subset of parameters. A class having n sets of fields which may or may not be initialized by the class user requires at most 2^n constructors for initializing these sets of fields.

For example, the `TextArea` class has three sets of fields (`rows`/`columns`, `text` and `scrollbars`), each of which may or may not be provided values for, at the moment of object creation. This means that the class designer will be forced to implement at most $2^3 = 8$ constructors.

Requirement 7 (Optional initialization)**.** If a class has n sets of fields which may or may not be initialized by the class user, we want our solution to allow the class designer to implement only $O(n)$ initializers.

Figure 1.2 shows how the `TextArea` class could be initialized in Pharo. Note that the class designer decided here to provide only five of the eight initialization options. The four instance-creation methods shown in Figure 1.2 are to be combined with the `initialize` method shown in Section 1.1.1. The five provided initialization options coincide with the following ways of creating text areas:

```
textArea1 := TextArea new.
textArea2 := TextArea rows: 5 columns: 50.
textArea3 := TextArea text: 'some text'.
textArea4 := TextArea text: 'some text' rows: 5 columns: 50.
textArea5 := TextArea text: 'some text' rows: 5 columns: 50 scrollbars: Scrollbars verticalOnly
```

There are three initialization options that have not been supported by the class designer. As a result, initializing a text area in one of these three legal ways requires additional work for the class user. For example, if the class user wants to create a text area with only the vertical scrollbar, and default values for all the other fields, two alternatives do exist:

```
Object subclass: #Point2D
  instanceVariableNames: 'x y'

Point2D class>>x: anInt y: anotherInt
  ^ self new
      x: anInt;
      y: anotherInt;
      yourself

Point2D class>>angle: angle rad: rad
  ^ self new [...]

Point2D class>>complex: aComplex
  ^ self new [...]

Point2D subclass: #ColoredPoint
  instanceVariableNames: 'r g b'

ColoredPoint class>>x: x y: y r: red g: green b: blue
  ^ (self x: x y: y)
       r: red;
       g: green;
       b: blue;
       yourself

ColoredPoint class>>angle: angle rad: rad
                    r: red g: green b: blue
  ^ (self angle: angle rad: rad)
       r: red;
       g: green;
       b: blue;
       yourself

ColoredPoint class>>x: x y: y c: c m: m yc: yc k: k
  ^ (self x: x y: y)
       r: 255 * (1 - c) * (1 - k);
       g: 255 * (1 - m) * (1 - k);
       b: 255 * (1 - yc) * (1 - k);
       yourself

[ ... 3 more instance-creation methods required to handle
  complex coordinates, and CMYK colors... ]
```

Figure 1.1: `Point2D` and `ColoredPoint` classes implemented in Pharo. Six instance-creation methods, with code duplication, must be implemented.

- the class user can use a provided constructor that matches a superset of the missing one, by providing values even for the fields whose default value is satisfactory; in the example, such an initializer is the following:

```
textArea6 := TextArea text: '' rows: 0 columns: 0 scrollbars: Scrollbars verticalOnly
```

- the class user can create a text area with all the default values, and then manually set the ones that are required to have a different value; in the example:

```
textArea6 := TextArea new scrollbars: Scrollbars verticalOnly
```

Implementing one instance-creation method for each legal subset of parameters requires code duplication. For example, the last three instance-creation methods in Figure 1.2 duplicate the definition of the `text` parameter. We refer the reader to Section 6.2 for a discussion on languages which contain named parameters or default parameter values, which partially solve this problem.

Finally, note that if the `TextArea` class designer does not want the `rows` and `columns` fields to be always initialized together, then the number of constructors to be implemented becomes at most $2^4 = 16$.

1.2.3 Problems with subclassing

When defining a subclass in Pharo, the instance-creation methods of the superclass are inherited and no work is required from the developer. However, some languages require additional work from the class designer. For example, Java requires a subclass designer to redefine the constructors of the superclass when they still represent valid ways to initialize the subclass (*i.e.,* the constructors of the superclass are not inherited).

Requirement 8 (Inheritance of the initialization protocol (no signature duplication))**.** We want our solution to allow a subclass designer to maintain the initialization protocol of the superclass without any need of signature duplications.

In both Java and Pharo though, when a subclass adds at least one additional field, the constructors of the superclass must be rewritten in the subclass. Let us consider a subclass of the previously mentioned `TextArea` class, which adds a boolean field to indicate whether the text area is enabled. This subclass must duplicate the five instance-creation methods of `TextArea`, each of which will begin with the same parameters of the corresponding superclass instance-creation method, adding a new one for initializing the new field.

Requirement 9 (Extension of the initialization protocol (no code duplication))**.** We want our solution to allow a subclass designer to extend the initialization protocol of the superclass without any need of code duplication.

```
Object subclass: #TextArea
  instanceVariableNames: 'rows columns text scrollbars'

TextArea class>>rows: anInt1 columns: anInt2
  ^ self new
      rows: anInt1;
      columns: anInt2;
      yourself

TextArea class>>text: aString
  ^ self new
      text: aString;
      yourself

TextArea class>>text: aString rows: anInt1 columns: anInt2
  ^ self new
      text: aString;
      rows: anInt1;
      columns: anInt2;
      yourself

TextArea class>>text: aString rows: anInt1 columns: anInt2
                scrollbars: aSymbol
  ^ self new
      text: aString;
      rows: anInt1;
      columns: anInt2;
      scrollbars: aSymbol;
      yourself
```

Figure 1.2: The `TextArea` class in Pharo. 7+1 instance-creation methods would be required to provide all the initialization options.

Chapter 2

The ini-module model

We now present a model for a Modular Initialization Protocol (*MIP*) that we have designed for Pharo. This could be seen also as a starting point for applying MIP to other dynamically-typed languages.

2.1 What is an ini-module and some definitions

Ini-modules were introduced in Magda [13, 4]. Magda is a mixin-oriented language designed with software reuse in mind: ini-modules are one answer to the problem of modularizing the initialization code. In Magda, the initialization protocol of an object is specified by writing small pieces of code, the ini-modules. Each ini-module may initialize part of an object or it may provide some data for other ini-modules, which will be executed afterwards.

Each ini-module is equipped with a (possibly empty) set of formal parameters, called *input parameters*, which can be supplied by the object-creation expression. An ini-module also specifies a (possibly empty) set of *output parameters*, referring by name to input parameters declared in other ini-modules, that will be computed by the ini-module and supplied to such other ini-modules. On the basis of the parameters supplied by the object-creation expression, some ini-modules rather than others will be executed.

We can distinguish four kinds of ini-modules: (1) if input and output parameters are declared, then the ini-module will typically use the input parameters to produce the output ones, *i.e.,* the ini-module may carry out a task of conversion of values; moreover, one or more fields may be initialized as well; (2) if no output parameters are declared but there are input ones, then the ini-module will use the input parameters to initialize one or more fields; (3) if no input parameters are declared but there are output ones, then the ini-module may compute default values for the output parameters; (4) finally, a special case of ini-module is the one with no parameters at all.

An example of ini-module of the fourth kind is the following.

Definition 1 (Root of class hierarchy)**.** Let `Object` be the root of the class hierarchy. We assume that there is always at least one ini-module in `Object`, that has no input parameters nor output parameters.

In the following we will use these notations:

- $\overline{a}$ to denote a *set* of elements $\{a_1, \ldots, a_n\}$.
- $\overrightarrow{a}$ to denote a *sequence* of elements $(a_1, \ldots, a_n)$. We assume that every sequence can be implicitly converted to a corresponding set.
- $\overrightarrow{a}[i]$ to denote the i-th element of the sequence $\overrightarrow{a}$, by indexing elements starting from 1.
- ε to denote the empty sequence.
- $\blacksquare$ to denote a runtime error.
- $A \cdot B$ to denote a *concatenation* of two sequences. In contexts in which a is an element, we will use $A \cdot a$ to denote $A \cdot (a)$, and $a \cdot A$ instead of $(a) \cdot A$. Moreover, we define $A \cdot \blacksquare = \blacksquare$.

The following is a semi-formal syntax of an ini-module abstracted away from any specific language:

optReq (`in_param_id`) `initializes` (`out_param_id`)
  `local_var`
  `I1`
  `next_ini`[`out_param_id` := `expr`];
  `I2`

- *optReq* stands for either `required` or `optional`, depending on whether the execution of the ini-module is mandatory or not; this feature may allow the class designer to enforce a complete initialization of the state of an object, by assigning each and every field with a value into at least one `required` ini-module;
- `in_param_id` is the (possibly empty) set of input parameter identifiers;
- `out_param_id` is the (possibly empty) set of output parameter identifiers whose values are computed by the ini-module;
- `local_var` are local variable declarations;
- `I1` (respectively `I2`) is a sequence of instructions, possibly empty, executed before (respectively after) the invocation of other ini-modules; `I1` and `I2` can both contain field assignments but not field accesses (see Check 7 later);
- `next_ini`[`out_param_id` := `expr`] is an assignment to the output parameters and an activation of other ini-modules, which the output parameters will be supplied to as input parameters.

The pseudo-syntax of an object-creation expression is:

$$\texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$$

where `C` is the class to be instantiated and $\overline{\texttt{id} := \texttt{expr}}$ are the initialization parameters together with their initialization expressions.

Consider a class `Point2D`, which is represented by its coordinates (`x`, `y`). We want the coordinates to be initialized in 3 different ways: (1) by providing Cartesian coordinates directly, (2) by providing polar coordinates, and (3) from another `Point2D` object.

Here are three ini-modules to initialize the coordinates of a `Point2D` in the three desired ways:

```
optional (angle, rad) initializes (coordX, coordY)
  next_ini[coordX := cos(angle) * rad, coordY := sin(angle) * rad];
// converts from polar coordinates to Cartesian coordinates

optional (point) initializes (coordX, coordY)
  next_ini[coordX := point.x, coordY := point.y];
// converts from another Point2D object to Cartesian coordinates

required (coordX, coordY) initializes ()
  x := coordX;
  y := coordY;
  next_ini[];
// initializes from Cartesian coordinates
```

The first two ini-modules carry out a task of conversion of values, respectively from polar coordinates and from another `Point2D` object to Cartesian coordinates: the ini-modules do so by outputting values that will be taken in input by the third ini-module. Such conversion is not mandatory, as it is necessary only if the class user does not provide directly Cartesian coordinates. Therefore we declared these ini-modules as `optional`. The third ini-module is responsible for initializing the `x` and `y` fields to values specified by the object-creation expression or by another ini-module that outputs them, such as one of the first two. This ini-module must be always executed (it is the one initializing the fields), thus we declared it as `required`.

With these three ini-modules, an object-creation expression can only specify either both `angle` and `rad`, only `point`, or both `coordX` and `coordY`:

```
new Point2D[angle := 5, rad := 20]
// the first and third ini-modules are executed

new Point2D[point := aPoint]
// the second and third ini-modules are executed

new Point2D[coordX := 5, coordY := 20]
// only the third ini-module is executed
```

Before showing how an object-creation expression is evaluated, we introduce some definitions.

Definition 2 (Ini-module signature)**.** We call *signature* of an ini-module the tuple consisting of its definition class, and its input and output parameter identifiers. Hereinafter, we will use the following notation to refer to an ini-module: `TheClass»(in_param_id)(out_param_id)`. We will sometimes omit the class when it is easily deducible from the context.

For example, to refer to the ini-module initializing a point's coordinates from another point, we write `Point2D»(point)(coordX, coordY)`. We can abbreviate the previous notation in `(point)(coordX, coordY)`, if it is obvious that we are talking about an ini-module of the class `Point2D`.

Constraint 1 (Ini-module signature uniqueness)**.** The signature of an ini-module uniquely identifies it.

This means that it is not possible to have, in the same class, two ini-modules with same input and output parameter identifiers. Note that it is conversely possible to have in the same class:

- *Two or more ini-modules with the same input parameters.* For example, a parameter can be used for calculating another parameter before being assigned to a field. Consider for example a class `Item` representing products in a catalog, with two fields, a `name` and an `id`. If we want to automatically generate an `id` for each item from its name, we can obtain the desired behaviour with the following set of ini-modules:

```
required (aName) initializes (aName, anId)
  next_ini[aName := aName, anId := //...generate an id...];
// automatically generates an id for the item

required (aName) initializes ()
  name := aName;
  next_ini[];
// initializes the item name

required (anId) initializes ()
  id := anId;
  next_ini[];
// initializes the item identifier
```

  Note that the first two ini-modules have both `aName` as the only input parameter. Another example can be splitting on different ini-modules the production of different output parameters from the same input parameters, *e.g.,* because not all of such output parameters necessarily must be produced. We will consider an example presenting such a situation in Figure 3.3.

- *One or more parameters that are both input and output parameters.* An ini-module can output one or more of its input parameters to make them available again to the next ini-modules; this can be useful in situations in which the same parameter must be used by different ini-modules. In the previous example of the class `Item`, the ini-module `(aName)(aName, anId)` uses the parameter `aName` to compute a value for `anId`; the parameter `aName` is outputted by it so that the ini-module `(aName)()` can set the

$$IModules(\texttt{Object}) = \varepsilon \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overrightarrow{E\ f}; \overrightarrow{IM}; \overrightarrow{M}\}}{IModules(C) = \overrightarrow{IM} \cdot IModules(D)}$$

Figure 2.1: Definition of function *IModules*.

field. An ini-module can output again a value it takes as input for different reasons. In particular, such an ini-module can:

- change the value (*e.g.,* from float to integer, from negative to positive);
- do checks on the value, raise a runtime error when incorrect, and return the value if everything is ok;
- use the value to produce something else (*e.g.,* `anId` in previous example) but without consuming the value.

We have just seen that the parameters of an ini-module can be both input and output parameters, in addition to being only one of the two. We introduce now some related definitions.

Definition 3 (Consumed parameter)**.** We say that a parameter `p` is *consumed* by an ini-module I if and only if it is input parameter of I but not output parameter of I.

For example, `aName` is a consumed by the the ini-module `Item»(aName)()` and `anId` is consumed by the ini-module `Item»(anId)()`.

Definition 4 (Produced parameter)**.** We say that a parameter `p` is *produced* by an ini-module I if and only if it is output parameter of I but not input parameter of I.

For example, `anId` is produced by the ini-module `Item»(aName)(aName, anId)`.

Definition 5 (Returning parameter)**.** We say that a parameter `p` is *returning* by an ini-module I if and only if it is simultaneously input and output parameter of I.

For example, `aName` is returning by the ini-module `Item»(aName)(aName, anId)`.

2.2 A semantics for ini-module activation

We describe the semantics of an object-creation expression by showing how the ini-modules defined in the hierarchy of a class are activated. We focus on how the ini-modules are activated based on the parameters submitted to the object-creation expression. This semantics is an adaptation of the one of Magda [4]. Note that here we will abstract away from the semantics of instructions and expressions contained in the body of an ini-module, *i.e.,* from the semantics of the instructions `I1` and `I2` and of the expressions $\overline{\texttt{expr}}$ in the instruction `next_ini` and in `new`.

$$RModules(C) = RModules(IModules(C))$$

$$RModules(\varepsilon) = \emptyset$$

$$\frac{optMod = \texttt{optional}\ (\overrightarrow{q})\ \texttt{initializes}\ (\overrightarrow{r})\ \{\dots\}}{RModules(optMod \cdot \overrightarrow{\texttt{mod}}) = RModules(\overrightarrow{\texttt{mod}})}$$

$$\frac{reqMod = \texttt{required}\ (\overrightarrow{q})\ \texttt{initializes}\ (\overrightarrow{r})\ \{\dots\}}{RModules(reqMod \cdot \overrightarrow{\texttt{mod}}) = reqMod \cup RModules(\overrightarrow{\texttt{mod}})}$$

Figure 2.2: Definition of function *RModules*.

$$activatedIniModules(C, \overline{p}) = activated(IModules(C), size(IModules(C)), \overline{p})$$

Figure 2.3: Definition of function *activatedIniModules*.

Figure 2.1 shows the definition of a function *IModules*. This function starts from the class C and goes up considering class by class in the hierarchy, collecting the ini-modules. We assume that there exists a total order among the ini-modules in a given class (note in the figure that indeed $\overrightarrow{IM}$ is a sequence and not a set). We justify these design choices in Section 2.5.1 and we discuss how to impose a total order among the ini-modules of a class in Chapter 3.

Figure 2.2 shows the definition of a function *RModules*. This function, when applied to a class C, collects all the `required` ini-modules defined through the hierarchy of C. With $RModules(\overrightarrow{\texttt{mod}}, i)$ we denote the result of *RModules* by restricting $\overrightarrow{\texttt{mod}}$ to the first i ini-modules.

Figure 2.3 shows the definition of a function *activatedIniModules*. This function computes a sequence of ini-modules that will be executed (we will define later a concept of *activable* ini-module), given a class C and a set of parameters $\overline{p}$. Given an object-creation expression

$$\texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$$

we want to describe its semantics by showing how the ini-modules activation process takes place. Which ini-modules get activated depends on (1) the parameters supplied in the object-creation expression, and (2) the ini-modules defined in the hierarchy of class `C`.

The semantics of the given object-creation expression is thus described by the call $activatedIniModules(\texttt{C}, \overline{\texttt{id}})$.

Definition 6 (Parameter map)**.** We call *parameter map* a map $< \overline{p}, \overline{v} >$ where $\overline{p}$ is a parameter set and $\overline{v}$ is the set of the respective values. This map is initially populated from the parameters supplied in an object-creation expression $\texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$: the keys are the $\overline{\texttt{id}}$ and the values are obtained by evaluating the corresponding $\overline{\texttt{expr}}$. Hereinafter we will sometimes refer only to the parameter set $\overline{p}$, when not interested in the values.

Figure 2.3 shows that the function *activatedIniModules* is defined as a call to another function *activated*. This function has three parameters: (1) a totally-ordered sequence of ini-modules to be considered for activation; (2) the step at which to stop the activation process (its use will be clear in Section 2.3); (3) the parameter set to start with the activation process.

The function *activated* is defined in Figure 2.4 through a recursive function *activated′* that has one more parameter for the current step i, which represents the index of the current ini-module in $\overrightarrow{\texttt{mod}}$ to be considered for activation. This function is based on five rules: each execution step considers the current ini-module $\overrightarrow{\texttt{mod}}[i]$ in the sequence, and at each computation step one of the rules is applied based on whether the current ini-module is activable or not:

Definition 7 (Activable ini-module)**.** Given an ini-module

$$\texttt{mod} = \mathit{optReq}\ (\overrightarrow{q})\ \texttt{initializes}\ (\overrightarrow{r})\ \{\dots\}$$

and a current set of parameters $\overline{p}$ to use for activation, we say that `mod` is *activable* if and only if both $\overrightarrow{q} \subseteq \overline{p}$ (*i.e.,* all the input parameters of `mod` are present in $\overline{p}$) and $(\overrightarrow{r} \setminus \overrightarrow{q}) \cap \overline{p} = \emptyset$ (*i.e.,* none of the produced parameters of `mod` are present in $\overline{p}$). We say that `mod` is *not activable* otherwise.

The second condition requires that, in order to activate an ini-module, none of its *produced* parameters (*i.e.,* given as output but not taken as input, based on Definition 4) is already present in the parameter map. We will see later that such a condition is needed to manage ini-modules producing default values for some fields.

The five rules defining the function *activated′* are triggered in the following situations:

- rule ⌊ACTIVATE⌋ is triggered if the *i-th* ini-module is activable: in this case the set of parameters $\overline{p}$ is updated by removing the input parameters and by adding the output parameters of the activated ini-module;
- rule ⌊NOTACTIVATEOPT⌋ is triggered if the *i-th* ini-module is not activable and it is declared as `optional`: in this case the set of parameters $\overline{p}$ does not change;
- rule ⌊NOTACTIVATEREQ⌋ is triggered if the *i-th* ini-module is not activable but it is declared as `required`;
- rule ⌊ENDCONDITION⌋ is triggered when the ini-modules to be considered for activation are completed and the parameter map is empty;
- rule ⌊OVERSUPPLIEDPARAMS⌋ is triggered when the ini-modules to be considered for activation are completed but there are still parameters into the parameter map.

We now show step-by-step on an example how the ini-modules get activated during the evaluation of an object-creation expression.

$$activated(\overrightarrow{\texttt{mod}}, s, \overline{p}) = activated'(\overrightarrow{\texttt{mod}}, 1, s, \overline{p}) \quad \text{with } 0 \leq s \leq |\overrightarrow{\texttt{mod}}|$$

$$\frac{s = 0 \qquad \vee \qquad (i > s \quad \wedge \quad \overline{p} = \emptyset)}{activated'(\overrightarrow{\texttt{mod}}, i, s, \overline{p}) = \varepsilon} \lfloor \textsc{EndCondition} \rfloor$$

$$\frac{s \neq 0 \qquad i > s \qquad \overline{p} \neq \emptyset}{activated'(\overrightarrow{\texttt{mod}}, i, s, \overline{p}) = \blacksquare} \lfloor \textsc{OversuppliedParams} \rfloor$$

$$\frac{\begin{array}{c}\overrightarrow{\texttt{mod}}[i] = optReq\ (\overrightarrow{q})\ \texttt{initializes}\ (\overrightarrow{r})\ \{\dots\} \\ 1 \leq i \leq s \qquad \overrightarrow{q} \subseteq \overline{p} \wedge (\overrightarrow{r} \setminus \overrightarrow{q}) \cap \overline{p} = \emptyset\end{array}}{activated'(\overrightarrow{\texttt{mod}}, i, s, \overline{p}) = \overrightarrow{\texttt{mod}}[i] \cdot activated'(\overrightarrow{\texttt{mod}}, i + 1, s, (\overline{p} \setminus \overrightarrow{q}) \cup \overrightarrow{r})} \lfloor \textsc{Activate} \rfloor$$

$$\frac{\begin{array}{c}\overrightarrow{\texttt{mod}}[i] = \texttt{optional}\ (\overrightarrow{q})\ \texttt{initializes}\ (\overrightarrow{r})\ \{\dots\} \\ 1 \leq i \leq s \qquad \overrightarrow{q} \not\subseteq \overline{p} \vee (\overrightarrow{r} \setminus \overrightarrow{q}) \cap \overline{p} \neq \emptyset\end{array}}{activated'(\overrightarrow{\texttt{mod}}, i, s, \overline{p}) = activated'(\overrightarrow{\texttt{mod}}, i + 1, s, \overline{p})} \lfloor \textsc{NotActivateOpt} \rfloor$$

$$\frac{\begin{array}{c}\overrightarrow{\texttt{mod}}[i] = \texttt{required}\ (\overrightarrow{q})\ \texttt{initializes}\ (\overrightarrow{r})\ \{\dots\} \\ 1 \leq i \leq s \qquad \overrightarrow{q} \not\subseteq \overline{p} \vee (\overrightarrow{r} \setminus \overrightarrow{q}) \cap \overline{p} \neq \emptyset\end{array}}{activated'(\overrightarrow{\texttt{mod}}, i, s, \overline{p}) = \blacksquare} \lfloor \textsc{NotActivateReq} \rfloor$$

Figure 2.4: Definition of function *activated*.

Consider a `Rectangle2D` class, whose instances are represented by the coordinates of its lower-left corner (`x`, `y`), a `width` and a `height`. Figure 2.5 shows a set of ini-modules for the class `Rectangle2D`. The ini-modules for initializing `x` and `y` are the same we previously defined for the class `Point2D` (lines 1, 5 and 13). In addition we define: an ini-module producing default values for the lower-left corner (line 9), assuming we want to place a rectangle by default in the origin; two ini-modules for initializing `width` and `height` (lines 19 and 24).

We also extend the class with a subclass `ColoredRectangle2D`, which adds three new fields `r`, `g` and `b` for color. To manage two different color palettes (CMYK and RGB), we only need to add two new ini-modules, as shown in Figure 2.6. The first ini-module calculates the values to be assigned to its output parameters by starting from its input parameters. The second ini-module takes three input parameters and simply assigns them to the newly introduced fields.

Imagine now we want to create a new colored rectangle (instance of the class `Colored-Rectangle2D`) as shown in Figure 2.7.

The semantics of this object-creation expression is described by the call:

$$
\begin{aligned}
&\mathit{activatedIniModules}(\texttt{ColoredRectangle2D}, \{\texttt{point}, \texttt{width}, \texttt{height}, \texttt{c}, \texttt{m}, \texttt{yc}, \texttt{k}\}) = \\
&\mathit{activated}(\mathit{IModules}(\texttt{ColoredRectangle2D}), \\
&\qquad 9, \\
&\qquad \{\texttt{point}, \texttt{width}, \texttt{height}, \texttt{c}, \texttt{m}, \texttt{yc}, \texttt{k}\}) = \\
&\mathit{activated}'(\mathit{IModules}(\texttt{ColoredRectangle2D}), \\
&\qquad 1, \\
&\qquad 9, \\
&\qquad \{\texttt{point}, \texttt{width}, \texttt{height}, \texttt{c}, \texttt{m}, \texttt{yc}, \texttt{k}\})
\end{aligned}
$$

where we suppose that:

$$
\begin{aligned}
&\mathit{IModules}(\texttt{ColoredRectangle2D}) = \\
&\qquad \texttt{ColoredRectangle2D»(c m yc k)(red green blue)} \\
&\qquad \cdot\ \texttt{ColoredRectangle2D»(red green blue)()} \\
&\qquad \cdot\ \texttt{Rectangle2D»(angle rad)(coordX coordY)} \\
&\qquad \cdot\ \texttt{Rectangle2D»(point)(coordX coordY)} \\
&\qquad \cdot\ \texttt{Rectangle2D»()(coordX coordY)} \\
&\qquad \cdot\ \texttt{Rectangle2D»(coordX coordY)()} \\
&\qquad \cdot\ \texttt{Rectangle2D»(width)()} \\
&\qquad \cdot\ \texttt{Rectangle2D»(height)()} \\
&\qquad \cdot\ \texttt{Object»()()}
\end{aligned}
$$

We show how the ini-modules defined in the instantiated class `ColoredRectangle2D` and its superclasses are activated based on the rules that define the function *activated′* (Figure 2.4). The computation of the function *activated′* is shown step-by-step in Figure 2.8. The execution flow is depicted in Figure 2.9, in which each box represents an ini-module, numbered with the step at which it is considered during the computation of *activated′*. At the beginning, the

```
optional (angle, rad) initializes (coordX, coordY)
  next_ini[coordX := cos(angle) * rad, coordY := sin(angle) * rad];
  // initializes lower-left corner from polar coordinates

optional (point) initializes (coordX, coordY)
  next_ini[coordX := point.x, coordY := point.y];
  // initializes lower-left corner from a Point object

optional () initializes (coordX, coordY)
  next_ini[coordX := 0, coordY := 0];
  // produces the default values for lower-left corner

required (coordX, coordY) initializes ()
  x := coordX;
  y := coordY;
  next_ini[];
  // initializes lower-left corner from Cartesian coordinates

required (aWidth) initializes ()
  width := aWidth;
  next_ini[];
  // initializes the width

required (aHeight) initializes ()
  height := aHeight;
  next_ini[];
  // initializes the height
```

Figure 2.5: The ini-modules defined for the class `Rectangle2D`.

```
optional (c, m, yc, k) initializes (red, green, blue)
  next_ini[red   := 255 * (1 - c)  * (1 - k),
           green := 255 * (1 - m)  * (1 - k),
           blue  := 255 * (1 - yc) * (1 - k)];
  // initializes color from a CMYK palette

required (red, green, blue) initializes ()
  r := red;
  g := green;
  b := blue;
  next_ini[];
  // initializes color from a RGB palette
```

Figure 2.6: The ini-modules defined for the class `ColoredRectangle2D`.

```
new ColoredRectangle2D[point := new Point2D(5,20),
                       width := 50,
                       height := 10,
                       c := 0.5,
                       m := 0.8,
                       yc := 0.3,
                       k := 0.1]
```

Figure 2.7: Object-creation expression for a new purple rectangle positioned in $(5, 20)$ of size 50×10.

$$
\begin{aligned}
& activated'(\overrightarrow{\texttt{mod}}, 1, 9, \{\texttt{point, width, height, c, m, yc, k}\}) \\
\lfloor \textsc{Activate} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot activated'(\overrightarrow{\texttt{mod}}, 2, 9, \{\texttt{point, width, height, red, green, blue}\}) \\
\lfloor \textsc{Activate} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot activated'(\overrightarrow{\texttt{mod}}, 3, 9, \{\texttt{point, width, height}\}) \\
\lfloor \textsc{NotActivateOpt} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot activated'(\overrightarrow{\texttt{mod}}, 4, 9, \{\texttt{point, width, height}\}) \\
\lfloor \textsc{Activate} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot \texttt{(point)(coordX coordY)} \cdot activated'(\overrightarrow{\texttt{mod}}, 5, 9, \{\texttt{width, height, coordX, coordY}\}) \\
\lfloor \textsc{NotActivateOpt} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot \texttt{(point)(coordX coordY)} \cdot activated'(\overrightarrow{\texttt{mod}}, 6, 9, \{\texttt{width, height, coordX, coordY}\}) \\
\lfloor \textsc{Activate} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot \texttt{(point)(coordX coordY)} \cdot \texttt{(coordX coordY)()} \cdot activated'(\overrightarrow{\texttt{mod}}, 7, 9, \{\texttt{width, height}\}) \\
\lfloor \textsc{Activate} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot \texttt{(point)(coordX coordY)} \cdot \texttt{(coordX coordY)()} \cdot \texttt{(width)()} \cdot activated'(\overrightarrow{\texttt{mod}}, 8, 9, \texttt{height}\}) \\
\lfloor \textsc{Activate} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot \texttt{(point)(coordX coordY)} \cdot \texttt{(coordX coordY)()} \cdot \texttt{(width)()} \cdot \texttt{(height)()} \cdot activated'(\overrightarrow{\texttt{mod}}, 9, 9, \emptyset\}) \\
\lfloor \textsc{Activate} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot \texttt{(point)(coordX coordY)} \cdot \texttt{(coordX coordY)()} \cdot \texttt{(width)()} \cdot \texttt{(height)()} \cdot \texttt{()()} \cdot activated'(\overrightarrow{\texttt{mod}}, 10, 9, \emptyset\}) \\
\lfloor \textsc{EndCondition} \rfloor \quad = \quad & \texttt{(c m yc k)(red green blue)} \cdot \texttt{(red green blue)()} \cdot \texttt{(point)(coordX coordY)} \cdot \texttt{(coordX coordY)()} \cdot \texttt{(width)()} \cdot \texttt{(height)()} \cdot \texttt{()()}
\end{aligned}
$$

Figure 2.8: Computation of the function *activated′* on the running example.

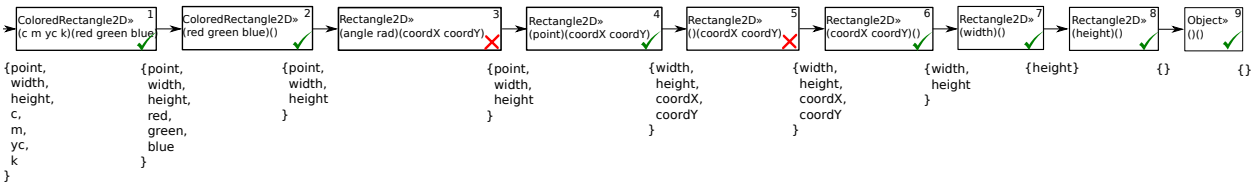

Figure 2.9: Execution flow of the object creation.

parameter set $\overline{p}$ contains all the parameters that have been submitted in the object-creation expression, *i.e.,* {`point`, `width`, `height`, `c`, `m`, `yc`, `k`}, as shown on the left of Figure 2.9.

Note that Figure 2.9 represents the ini-modules of the `ColoredRectangle2D` class hierarchy as a sequential chain because none of the ini-modules defined in the hierarchy defines a sequence of instructions `I2`. We will return on what happens when also `I2` instructions are defined in Section 2.4.

We now comment each step of the computation:

1. `ColoredRectangle2D»(c m yc k)(red green blue)` is activated by rule ⌊ACTIVATE⌋ because its input parameters `c`, `m`, `yc` and `k` are all in the parameter set $\overline{p}$, while its produced parameters `red`, `green` and `blue` are not in $\overline{p}$; the parameter set is then updated accordingly to the rule, removing the input parameters of the activated ini-module and adding its output parameters;

2. `ColoredRectangle2D»(red green blue)()` is activated by rule ⌊ACTIVATE⌋ because now we have `red`, `green` and `blue` in $\overline{p}$, and the ini-module produces no parameters; the three input parameters of the activated ini-module are removed from $\overline{p}$;

3. `Rectangle2D»(angle rad)(coordX coordY)` is not activated by rule ⌊NOTACTIVATEOPT⌋ because `angle` and `rad` are not in $\overline{p}$, which contains values for `point`, `width` and `height`;

4. `Rectangle2D»(point)(coordX coordY)` is activated by rule ⌊ACTIVATE⌋ because `point` is in $\overline{p}$ while the produced parameters `coordX` and `coordY` are not; $\overline{p}$ is updated by removing `point` and by adding `coordX` and `coordY`;

5. `Rectangle2D»()(coordX coordY)` is not activated by rule ⌊NOTACTIVATEOPT⌋ because its produced parameters `coordX` and `coordY` are already in $\overline{p}$; note that this is correct because we do not need default values for `coordX` and `coordY` as the values for coordinates have been submitted in the `new` (Figure 2.7);

6. `Rectangle2D»(coordX coordY)()` is activated by rule ⌊ACTIVATE⌋ and its two input parameters are removed from $\overline{p}$;

7. `Rectangle2D»(width)()` is activated by rule ⌊ACTIVATE⌋ and its input parameter is removed from $\overline{p}$;

8. `Rectangle2D»(height)()` is activated by rule ⌊ACTIVATE⌋ and its input parameter is removed from $\overline{p}$;

9. finally, `Object»()()` is activated by rule ⌊ACTIVATE⌋.

The computation is ended by application of rule ⌊ENDCONDITION⌋, obtaining the sequence of the ini-modules activated by the given object-creation expression (look at the last line of Figure 2.8).

We introduce now two lemmas and some definitions that will be useful in the following.

Lemma 1. Given $k < |\overrightarrow{\texttt{mod}}|$ we have that:

$$activated(\overrightarrow{\texttt{mod}}, k+1, \overline{p}) = activated(\overrightarrow{\texttt{mod}}, k, \overline{p}) \cdot activated(\overrightarrow{\texttt{mod}}[k+1], 1, \overline{p})$$

Proof. We prove the lemma by induction on the length of the ini-modules sequence $\overrightarrow{\texttt{mod}}$. If $|\overrightarrow{\texttt{mod}}| = 0$ then there are no legal values of k, since the side condition of the function *activated* requires $s \geq 0$. If $|\overrightarrow{\texttt{mod}}| = 1$, then we must prove that:

$$activated(\overrightarrow{\texttt{mod}}, 1, \overline{p}) = activated(\overrightarrow{\texttt{mod}}, 0, \overline{p}) \cdot activated(\overrightarrow{\texttt{mod}}[1], 1, \overline{p})$$

where $activated(\overrightarrow{\texttt{mod}}, 0, \overline{p})$ reduces to ε and $\overrightarrow{\texttt{mod}}[1] = \overrightarrow{\texttt{mod}}$ because by hypothesis $|\overrightarrow{\texttt{mod}}| = 1$. Therefore the equality is proved in the base case.

For the inductive case, we consider two subsequences of $\overrightarrow{\texttt{mod}}$, calling $\overrightarrow{\texttt{mod}_1}$ the sequence of the first j elements of $\overrightarrow{\texttt{mod}}$, and $\overrightarrow{\texttt{mod}_2}$ the sequence of the first $j+1$ elements of $\overrightarrow{\texttt{mod}}$. Then we assume the thesis to be true for a sequence of length j:

$$activated(\overrightarrow{\texttt{mod}_1}, k+1, \overline{p}) = activated(\overrightarrow{\texttt{mod}_1}, k, \overline{p}) \cdot activated(\overrightarrow{\texttt{mod}_1}[k+1], 1, \overline{p})$$

with $k < j$, and we prove that it is then true also for a sequence of length $j+1$:

$$activated(\overrightarrow{\texttt{mod}_2}, k+1, \overline{p}) = activated(\overrightarrow{\texttt{mod}_2}, k, \overline{p}) \cdot activated(\overrightarrow{\texttt{mod}_2}[k+1], 1, \overline{p})$$

with $k < j$. To do that we use the following property that follows from the definition of the function *activated*:

$$s \leq |\overrightarrow{\texttt{mod}_1}| \quad \Rightarrow \quad activated(\overrightarrow{\texttt{mod}_1}, s, \overline{p}) = activated(\overrightarrow{\texttt{mod}_2}, s, \overline{p})$$

By applying this property and the inductive hypothesis we have that:

$$\begin{aligned} activated(\overrightarrow{\texttt{mod}_2}, k+1, \overline{p}) &\overset{(*)}{=} activated(\overrightarrow{\texttt{mod}_1}, k+1, \overline{p}) \\ &\overset{\text{hyp.}}{=} activated(\overrightarrow{\texttt{mod}_1}, k, \overline{p}) \cdot activated(\overrightarrow{\texttt{mod}_1}[k+1], 1, \overline{p}) \\ &\overset{(**)}{=} activated(\overrightarrow{\texttt{mod}_2}, k, \overline{p}) \cdot activated(\overrightarrow{\texttt{mod}_2}[k+1], 1, \overline{p}) \end{aligned}$$

The first equality (*) is obtained by applying the above property, whose premise is satisfied because $k+1 \leq j$, which is true because by hypothesis $k < j$. The second equality is obtained by applying the inductive hypothesis. Finally, the third equality (**) is obtained by observing that $activated(\overrightarrow{\texttt{mod}_1}, k, \overline{p}) = activated(\overrightarrow{\texttt{mod}_2}, k, \overline{p})$ again for the above property, since $k \leq j$, and that $\overrightarrow{\texttt{mod}_1}[k+1] = \overrightarrow{\texttt{mod}_2}[k+1]$. □

Lemma 2 (Activation of `required` ini-modules)**.** If the result of the function $activated(\overrightarrow{\texttt{mod}}, s, \overline{p})$ is a sequence of ini-modules $\overrightarrow{\texttt{S}} \neq \blacksquare$, then $RModules(\overrightarrow{\texttt{mod}}, s) \subseteq \overrightarrow{\texttt{S}}$.

Proof. We prove the lemma by induction. If $s = 0$ then

$$activated(\overrightarrow{\texttt{mod}}, 0, \overline{p}) = activated'(\overrightarrow{\texttt{mod}}, 1, 0, \overline{p}) = \varepsilon$$

Moreover, $RModules(\overrightarrow{\texttt{mod}}, 0) = \emptyset$. Therefore the thesis is proved in the base case because $\emptyset \subseteq \varepsilon$. Assume now that if $activated(\overrightarrow{\texttt{mod}}, k, \overline{p}) = \overrightarrow{\texttt{S}_1}$, then $RModules(\overrightarrow{\texttt{mod}}, k) \subseteq \overrightarrow{\texttt{S}_1}$, with $\overrightarrow{\texttt{S}_1} \neq \blacksquare$. If we assume that $activated(\overrightarrow{\texttt{mod}}, k+1, \overline{p}) = \overrightarrow{\texttt{S}_2}$, with $\overrightarrow{\texttt{S}_2} \neq \blacksquare$, by applying Lemma 1 we can decompose it into:

$$activated(\overrightarrow{\texttt{mod}}, k+1, \overline{p}) = activated(\overrightarrow{\texttt{mod}}, k, \overline{p}) \cdot activated(\overrightarrow{\texttt{mod}}[k+1], 1, \overline{p})$$

Now, by hypothesis we have that $activated(\overrightarrow{\texttt{mod}}, k+1, \overline{p})$ is a sequence $\overrightarrow{\texttt{S}_2} \neq \blacksquare$. To have that we must have that $activated(\overrightarrow{\texttt{mod}}, k, \overline{p}) = \overrightarrow{\texttt{S}_1}$ and that $activated(\overrightarrow{\texttt{mod}}[k+1], 1, \overline{p}) = \overrightarrow{\texttt{S}_3}$, with $\overrightarrow{\texttt{S}_1} \neq \blacksquare$ and $\overrightarrow{\texttt{S}_3} \neq \blacksquare$.

As a consequence, from the inductive hypothesis we can conclude that $RModules(\overrightarrow{\texttt{mod}}, k) \subseteq \overrightarrow{\texttt{S}_1}$. To prove that $RModules(\overrightarrow{\texttt{mod}}, k+1) \subseteq \overrightarrow{\texttt{S}_2}$, since $\overrightarrow{\texttt{S}_2} = \overrightarrow{\texttt{S}_1} \cdot \overrightarrow{\texttt{S}_3}$, we must still prove only that $activated(\overrightarrow{\texttt{mod}}[k+1], 1, \overline{p})$ activates the ini-module $\overrightarrow{\texttt{mod}}[k+1]$, if it is `required`. Based on the function definition shown in Figure 2.4, we have that:

$$activated(\overrightarrow{\texttt{mod}}[k+1], 1, \overline{p}) = activated'(\overrightarrow{\texttt{mod}}[k+1], 1, 1, \overline{p})$$

At the first step of the computation of $activated'(\overrightarrow{\texttt{mod}}[k+1], 1, 1, \overline{p})$, we have that $i = 1$ and $s = 1$, therefore one of the following rules will be applied:

- if the rule $\lfloor$ActivateReq$\rfloor$ is applied then the thesis is verified, because $\overrightarrow{\texttt{mod}}[k+1]$ is queued to the sequence of activated ini-modules;
- if the rule $\lfloor$NotActivateOpt$\rfloor$ is applied then the thesis is verified, because $\overrightarrow{\texttt{mod}}[k+1]$ is not queued to the sequence of activated ini-modules, but it is `optional`.

Note that the rule $\lfloor$NotActivateReq$\rfloor$ can not be applied, because we must have that $activated(\overrightarrow{\texttt{mod}}[k+1], 1, \overline{p}) = \overrightarrow{\texttt{S}_3}$ with $\overrightarrow{\texttt{S}_3} \neq \blacksquare$.

At the next step of the computation of $activated'(\overrightarrow{\texttt{mod}}[k+1], 1, 1, \overline{p})$, we have that $i = 2$ and $s = 1$, therefore the rule $\lfloor$EndCondition$\rfloor$ will be applied. Indeed, the rule $\lfloor$OversuppliedParams$\rfloor$ can not be applied, again because we must have $activated(\overrightarrow{\texttt{mod}}[k+1], 1, \overline{p}) = \overrightarrow{\texttt{S}_3}$ with $\overrightarrow{\texttt{S}_3} \neq \blacksquare$. □

Definition 8 (New-supplied parameter)**.** Given an object-creation expression

$$\texttt{E} = \texttt{new C}[\texttt{id}_1 := \texttt{e}_1, \ldots, \texttt{id}_n := \texttt{e}_n]$$

we say that a parameter `p` is *new-supplied* in `E` (or that it is *supplied directly* in `E`) if and only if $\texttt{p} \in \overline{\texttt{id}}$.

For example, the parameters `point`, `width`, `height`, `c`, `m`, `yc` and `k` are all new-supplied in the object-creation expression of Figure 2.7.

Definition 9 (Module-supplied parameter)**.** Given an object-creation expression

$$\texttt{E} = \texttt{new C}[\overline{\texttt{id := expr}}]$$

we say that a parameter p is *module-supplied* in E (or that it is *supplied indirectly* in E) by an ini-module I if and only if I is activated by E and p is produced by I.

For example, the parameters coordX and coordY are module-supplied by the ini-module Rectangle2D»(point)(coordX coordY) in the object-creation expression of Figure 2.7. Indeed, coordX and coordY are produced by Rectangle2D»(point)(coordX coordY), which is activated by the given object-creation expression, as we have seen in the step-by-step execution.

Definition 10 (Supplied parameter)**.** Given an object-creation expression

$$\texttt{E} = \texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$$

we say that a parameter p is *supplied* if and only if it is new-supplied or module-supplied in E (*i.e.,* if and only if it is directly or indirectly supplied in E).

Let $\overrightarrow{\texttt{mod}} = \mathit{IModules}(\texttt{C})$. We denote with $\mathit{suppliedBy}(\overrightarrow{\texttt{mod}}, \overline{\texttt{id}})$ the set of the parameters $\overline{\texttt{p}}$ such that are supplied in E. Analogously, we denote with $\mathit{suppliedBy}(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})$ the set of the parameters $\overline{\texttt{p}}$ such that they are supplied in E, restricting $\overrightarrow{\texttt{mod}}$ to the first i ini-modules.

For example, both parameters point and coordX are supplied in the object-creation expression of Figure 2.7: the first one is supplied directly, while the second one is supplied indirectly.

Definition 11 (Terminated parameter)**.** Given an object-creation expression

$$\texttt{E} = \texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$$

we say that a parameter p is *terminated* in E by an ini-module I if and only if I is activated by E and p is consumed by I.

Let $\overrightarrow{\texttt{mod}} = \mathit{IModules}(\texttt{C})$. We denote with $\mathit{notTerminatedBy}(\overrightarrow{\texttt{mod}}, \overline{p})$ the set of the parameters $\overline{\texttt{p}}$ such that they are not terminated by any ini-module in E. Analogously, we denote with $\mathit{notTerminatedBy}(\overrightarrow{\texttt{mod}}, i, \overline{p})$ the set of the parameters $\overline{\texttt{p}}$ such that they are not terminated in E, restricting $\overrightarrow{\texttt{mod}}$ to the first i ini-modules.

For example, the parameters coordX and coordY are terminated by the ini-module Rectangle2D»(coordX coordY)() in the object-creation expression of Figure 2.7. Indeed, coordX and coordY are consumed by Rectangle2D»(coordX coordY)(), which is activated by the given object-creation expression, as we have seen in the step-by-step execution.

Definition 12 (Permanently terminated parameter)**.** Given an object-creation expression

$$\texttt{E} = \texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$$

we say that a parameter p is *permanently terminated* in E if and only if $\mathit{IModules}(\texttt{C})$ can be written as $\overrightarrow{\texttt{mod}_1} \cdot \texttt{I} \cdot \overrightarrow{\texttt{mod}_2}$, where I is an ini-module such that:

- p is terminated by I, *and*
- $\overrightarrow{\texttt{mod}_2}$ does not contain any ini-module supplying p in E.

Let $\overrightarrow{\texttt{mod}} = \mathit{IModules}(\texttt{C})$. We denote with $\mathit{notPermTerminatedBy}(\overrightarrow{\texttt{mod}}, \overline{p})$ the set of the parameters $\overline{\texttt{p}}$ such that they are not permanently terminated in E. Analogously, we denote with $\mathit{notPermTerminatedBy}(\overrightarrow{\texttt{mod}}, i, \overline{p})$ the set of the parameters $\overline{\texttt{p}}$ such that they are not permanently terminated in E, restricting $\overrightarrow{\texttt{mod}}$ to the first i ini-modules.

For example, all the parameters that are terminated in the object-creation expression of Figure 2.7 are also permanently terminated in it.

Note that a parameter p is not permanently terminated in an object-creation expression E when one of the two required conditions is false, *i.e.,* when (1) p is not terminated by any ini-module in E, or (2) p is terminated in E by an ini-module I, but p is module-supplied in E by an ini-module activated after I.

Case (1) is trivial, because if p is not terminated in E, then it is also not permanently terminated in E. Case (2) means that a situation like the following may arise in a class C:

```
optional () initializes (a)
  next_ini[a := aConst];

required (a) initializes ()
  a_field := a;
  next_ini[];

optional (b) initializes (a)
  next_ini[a := fun(b)];

required (a, c) initializes ()
  x_field := fun(a,c);
  next_ini[];
```

Indeed, in our model each output parameter of an ini-module must be an input parameter of *at least* one ini-module coming after in the order (see Check 2 later).

Consider an object-creation expression `new C[a := val1, b := val2, c := val3]` and suppose the ini-modules are ordered as they are written above. Parameter `a` is first new-supplied, then it is (not permanently!) terminated by ini-module `(a)()`, then it is supplied again by ini-module `(b)(a)`, and finally it gets permanently terminated by ini-module `(a, c)()`. Note that, even though they have the same name, the first parameter `a` (that is new-supplied, and terminated by `(a)()`) and the second parameter `a` (that is module-supplied by `(b)(a)`, and terminated by `(a, c)()`) are independent, because their "life cycles" do not overlap.

As a future work, we plan to prove that, in such cases, a renaming of the parameters is always possible. That is, for each parameter p, if p is terminated by an ini-module I in an object-creation expression E, then p is not module-supplied again in E by another ini-module $\texttt{I}'$ activated after I.

For instance, the above example could be rewritten as follows:

```
optional () initializes (a1)
  next_ini[a1 := aConst];

required (a1) initializes ()
  a_field := a1;
  next_ini[];
```

```
optional (b) initializes (a2)
  next_ini[a2 := fun(b)];

required (a2, c) initializes ()
  x_field := fun(a2,c);
  next_ini[];
```

In particular, we want to prove that the two following properties hold, *i.e.,* that such a renaming preserves:

1. the ordering in which ini-modules are considered for activation;
2. the semantics of the defined ini-modules, *i.e.,* the activated sequence of ini-modules $activatedIniModules(\texttt{C}, \overline{\texttt{id}})$ does not change.

Note that the first property is to be proved as a precondition for the second one. Indeed, looking at Figure 2.3 is possible to see that the function *activatedIniModules* depends on the function *IModules*, *i.e.,* the sequence of the activated ini-modules depends on the ordering in which ini-modules are considered for activation.

What follows was developed also to be able to prove renaming soundness according the two above properties.

Definition 13 (Oversupplied parameter)**.** Given an object-creation expression

$$\texttt{E} = \texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$$

and let $\overrightarrow{\texttt{mod}} = IModules(\texttt{C})$, we say that a parameter `p` is *oversupplied* in `E` if and only if:

$$\texttt{p} \in notPermTerminatedBy(\overrightarrow{\texttt{mod}}, suppliedBy(\overrightarrow{\texttt{mod}}, \overline{\texttt{id}}))$$

i.e., if and only if `p` is (directly or indirectly) supplied but not permanently terminated in $\texttt{E} = \texttt{new C}[\overline{\texttt{id} := \texttt{expr}}]$. We denote with $oversupplied(\overrightarrow{\texttt{mod}}, \overline{p})$ the set of the parameters $\overline{p}$ such that they are oversupplied in `E`. Analogously, we denote with $oversupplied(\overrightarrow{\texttt{mod}}, i, \overline{p})$ the set of the parameters $\overline{p}$ such that they are oversupplied in `E`, restricting $\overrightarrow{\texttt{mod}}$ to the first i ini-modules.

In the object-creation expression of Figure 2.7, none of the parameters is oversupplied. Indeed, all the parameters that are new-supplied or module-supplied in the object-creation expression get permanently terminated at the end of the ini-modules activation process. Consider another object-creation expression, supplying both parameters `point`, and `coordX`/`coordY`:

```
new ColoredRectangle2D[point := new Point2D(5,20),
                       coordX := 0,
                       coordY := 0,
                       ...]
```

This object-creation expression will cause the ini-module `Rectangle2D»(point)(coordX, coordY)` to not be activated. Indeed, the presence in the parameter map of its produced parameters `coordX` and `coordY` (which are new-supplied) prevents its activation. The ini-module `Rectangle2D»(coordX coordY)()` will then be activated, setting the fields `x` and `y` to the origin `(0,0)`. The new-supplied parameter `point` is thus oversupplied, because it does not get permanently terminated.

Algorithm 1 Evaluates an object-creation expression

Input: An object-creation expression `new C[`$\overline{\texttt{id := expr}}$`]` to be evaluated.
Output: A new initialized instance of the class `C`.

1: **function** CreateInstance(*exprNew*)
2: $\overline{\texttt{id := val}} \longleftarrow$ evaluate the $\overline{\texttt{expr}}$ in *exprNew*
3: *pm* $\longleftarrow$ a new map populated from $\overline{\texttt{id := val}}$
4: *o* $\longleftarrow$ create an uninitialized instance of `C`
5: $\overrightarrow{\texttt{mod}} \longleftarrow$ IniModulesInHierarchy(`C`)
6: `mod` $\longleftarrow$ remove the first ini-module from $\overrightarrow{\texttt{mod}}$
7: TryToActivate(`mod`, *o*, *pm*, $\overrightarrow{\texttt{mod}}$)
8: **return** *o*
9: **end function**

Algorithm 2 Retrieves the ini-modules through the hierarchy of a class `C`

Input: A class `C`.
Output: An ordered sequence of all the ini-modules through the hierarchy of `C`.

1: **function** IniModulesInHierarchy(`C`)
2: *c* $\longleftarrow$ `C`
3: $\overrightarrow{\texttt{mod}} \longleftarrow \overrightarrow{\texttt{mod}} \cdot$ IniModulesOf(*c*)
4: **while** *c* $\neq$ `Object` **do**
5: *c* $\longleftarrow$ superclass of `C`
6: $\overrightarrow{\texttt{mod}} \longleftarrow \overrightarrow{\texttt{mod}} \cdot$ IniModulesOf(*c*)
7: **end while**
8: **return** $\overrightarrow{\texttt{mod}}$
9: **end function**

2.3 The algorithm

We detail now an algorithm that implements the ini-module semantics introduced in the previous section.

We comment step-by-step the function CreateInstance, which is shown in Algorithm 1:

- the parameter map (*pm*) is populated starting from the parameters supplied in the object creation expression `new C[`$\overline{\texttt{id := expr}}$`]`: as stated by Definition 6, this means evaluating the corresponding $\overline{\texttt{expr}}$ (line 2), and then using the resulting $\overline{\texttt{id := val}}$ to populate the map, with the $\overline{\texttt{id}}$ as keys and the $\overline{\texttt{val}}$ as values (line 3).
- a new instance of class `C` is created (line 4);
- a sequence of ini-modules is extracted (line 5); the function IniModulesInHierarchy, which is shown in Algorithm 2, directly implements the function *IModules*, starting from the class `C` and going up in the class hierarchy; for each encountered class, the function IniModulesOf retrieves a totally ordered sequence of the ini-modules defined in the class `C` and it is detailed in Chapter 3;

- each ini-module is tried based on the current parameter map *pm*; the procedure TRYTOACTIVATE implements recursively the ini-modules activation process and it is started on the first ini-module in the sequence (lines 6-7);
- when the recursive process completes, the initialization process terminates returning the initialized object (line 8).

The recursive procedure TRYTOACTIVATE is shown in Algorithm 3. The procedure is executed on an ini-module `mod` with an object *o* to be initialized, a parameter map *pm* and a sequence of ini-modules $\overrightarrow{\texttt{mod}}$. Each execution of the procedure TRYTOACTIVATE determines if the current ini-module must be activated or not, and acts accordingly.

The if-branch of the procedure (lines 4-18) manages the activation of the ini-module `mod`. The ini-module activation is composed of three steps:

- the execution of the instructions `I1` in the body of the ini-module (line 6);
- the recursive call to resume the activation process on the next ini-module to be considered (lines 8-17), if there is one; note that, before calling recursively TRYTOACTIVATE, the input parameters of the activated ini-module are removed from the parameter map *pm* (line 10); likewise, its output parameters are added to the parameter map *pm* (line 11);
- the execution of the instructions `I2` in the body of the ini-module (line 18).

The else-branch of the procedure (lines 19-32) may reactivate the recursive process on the next ini-module, if `mod` must not be activated. Note that a runtime error is raised if a `required` ini-module is not activated.

We introduce now an invariant for the recursive procedure TRYTOACTIVATE that we use for proving the correctness of the algorithm. In the following, with *step i* we mean:

- if $i = 0$, the computation step preceding the first call to TRYTOACTIVATE;
- if $i > 0$, the *i-th* recursive call of TRYTOACTIVATE.

Definition 14 (Invariant)**.** Given an object-creation expression $\texttt{E} = \texttt{new C}\overline{[\texttt{id} := \texttt{expr}]}$ and let $\overrightarrow{\texttt{mod}} = \text{INIMODULESINHIERARCHY}(\texttt{C})$, at step *i* of the computation of TRYTOACTIVATE(`mod`, *o*, *pm*, $\overrightarrow{\texttt{mod}}$), the activated ini-modules are a sequence $\overrightarrow{\texttt{S}} \neq \blacksquare$ including all the `required` ini-modules that have been considered for activation *or* an error has been raised:

$$(activated(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}) = \overrightarrow{\texttt{S}} \ \wedge \ RModules(\overrightarrow{\texttt{mod}}, i) \subseteq \overrightarrow{\texttt{S}}) \ \vee \ activated(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}) = \blacksquare \quad (2.1)$$

Moreover, the parameter map contains a key for each of:

$$notPermTerminatedBy(\overrightarrow{\texttt{mod}}, i, suppliedBy(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})) \quad (2.2)$$

Proof. We can observe that if $activated(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}) = \overrightarrow{\texttt{S}}$, with $\overrightarrow{\texttt{S}} \neq \blacksquare$, then $RModules(\overrightarrow{\texttt{mod}}, i) \subseteq \overrightarrow{\texttt{S}}$ for Lemma 2. Therefore, 2.1 reduces to

Algorithm 3 Recursively implements the ini-modules activation process

Input: The current ini-module `mod` to be considered for activation, the object o to be initialized, the parameter map pm and the ini-modules sequence $\overrightarrow{\texttt{mod}}$.

```
1: procedure TryToActivate(mod, o, pm, mod⃗)
2:     Let Ip be the set of input parameter identifiers of mod
3:     Let NextInstruction be the instruction next_ini[out_param_id := expr] contained in the body of mod
4:     if mod is activable given pm then
5:         inputParams ⟵ retrieve in pm the actual values for the identifiers in Ip
6:         execute I1 using inputParams
7:         outputParams = evaluate the expressions in NextInstruction
8:         if mod⃗ is not empty then
9:             nextMod ⟵ remove the first ini-module from mod⃗
10:            remove all inputParams from pm
11:            add all outputParams to pm
12:            TryToActivate(nextMod, o, pm, mod⃗)
13:        else
14:            if pm is not empty then
15:                raise an error
16:            end if
17:        end if
18:        execute I2 using inputParams
19:    else
20:        if mod is required then
21:            raise an error
22:        else
23:            if mod⃗ is not empty then
24:                nextMod ⟵ remove the first ini-module from mod⃗
25:                TryToActivate(nextMod, o, pm, mod⃗)
26:            else
27:                if pm is not empty then
28:                    raise an error
29:                end if
30:            end if
31:        end if
32:    end if
33: end procedure
```

$$activated(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}) = \overrightarrow{\texttt{S}} \ \vee \ activated(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}) = \blacksquare$$

with $\overrightarrow{\texttt{S}} \neq \blacksquare$, *i.e.,* we must prove that the behaviour of the procedure tryToActivate coincides with that of $activated(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})$. We prove that by induction.

Base case. We must first prove that the invariant is true prior to the first call to the recursive procedure TryToActivate. At step 0 of the computation, the ini-modules activated should be those contained in:

$$\begin{aligned} & activated(\overrightarrow{\texttt{mod}}, 0, \overline{\texttt{id}}) \\ & = activated(\overrightarrow{\texttt{mod}}, 1, 0, \overline{\texttt{id}}) \\ & = \varepsilon \end{aligned}$$

where $\overrightarrow{\texttt{mod}}$ = IniModulesInHierarchy(C). Indeed, no ini-module is activated by the function CreateInstance prior to the first call to TryToActivate. Moreover, we have $RModules(\overrightarrow{\texttt{mod}}, 0) = \varepsilon$ and thus $\varepsilon \subseteq \varepsilon$.

The parameter map should contain a key for each of:

$$\begin{aligned} & notPermTerminatedBy(\overrightarrow{\texttt{mod}}, 0, suppliedBy(\overrightarrow{\texttt{mod}}, 0, \overline{\texttt{id}})) \\ & = notPermTerminatedBy(\overrightarrow{\texttt{mod}}, 0, \overline{\texttt{id}}) \\ & = \overline{\texttt{id}} \end{aligned}$$

Indeed, the parameter map is initialized with $\overline{\texttt{id} := \texttt{val}}$ (line 3 of Algorithm 1). Therefore, the invariant is verified in the initialization phase.

Inductive case. We must now prove that, if the invariant is true for step i, it is true for step $i + 1$. Therefore, if at step i the behaviour of the procedure tryToActivate is described by:

$$activated(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})$$

we must prove that at step $i + 1$ it is described by:

$$activated(\overrightarrow{\texttt{mod}}, i + 1, \overline{\texttt{id}})$$

where $\overrightarrow{\texttt{mod}}$ = IniModulesInHierarchy(C).

First of all, note that in Algorithm 1 $\overrightarrow{\texttt{mod}}$ is initialized with IniModulesInHierarchy(C) (line 5). Proving the above invariant corresponds to showing that the procedure TryToActivate computes the function *activated*.

We can observe that, in Algorithm 3, the most external if-branch implements the rule ⌊Activate⌋ (lines 4-18), while the else-branch implements the rules ⌊NotActivateReq⌋ (lines 20-21) and ⌊NotActivateOpt⌋ (lines 22-31). The parameter map *pm* of Algorithm 3 directly corresponds to the parameter set $\overline{p}$ of the function *activated*. The end-condition of the recursion, defined by the rule ⌊EndCondition⌋, is verified in both branches, before the recursive call (lines 8 and 23), while the corresponding else-branches (lines 13 and 26) verify the oversupplied parameters condition, defined by the rule ⌊OversuppliedParams⌋.

Moreover, if at step i the parameter map contains a key for each of:

$$notPermTerminatedBy(\overrightarrow{\texttt{mod}}, i, suppliedBy(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}))$$

we must prove that at step $i+1$ it will contain a key for each of:

$$notPermTerminatedBy(\overrightarrow{\texttt{mod}}, i+1, suppliedBy(\overrightarrow{\texttt{mod}}, i+1, \overline{\texttt{id}})) \qquad (*)$$

This means that, assuming that at step i we have in the parameter map all the parameters that are oversupplied based on the first i ini-modules in the sequence $\overrightarrow{\texttt{mod}}$, *i.e.,* $oversupplied(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})$, we must prove that at step $i+1$ the algorithm updates the parameter map in such a way it contains all the parameters that are oversupplied based on the first $i+1$ ini-modules in the sequence $\overrightarrow{\texttt{mod}}$, *i.e.,* $oversupplied(\overrightarrow{\texttt{mod}}, i+1, \overline{\texttt{id}})$.

We note that we can rewrite $suppliedBy(\overrightarrow{\texttt{mod}}, i+1, \overline{\texttt{id}})$ as follows, by adding and at the same time subtracting $suppliedBy(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})$ to it:

$$suppliedBy(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}) + (suppliedBy(\overrightarrow{\texttt{mod}}, i+1, \overline{\texttt{id}}) \setminus suppliedBy(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}}))$$

Consequently:

- the parameters in $suppliedBy(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})$ must continue to belong to *(*)*, unless they are terminated by the $(i+1)st$ ini-module; this means that *(a)* the parameters that are permanently terminated by the $(i+1)st$ ini-module must be removed from the parameter map.
- $suppliedBy(\overrightarrow{\texttt{mod}}, i+1, \overline{\texttt{id}}) \setminus suppliedBy(\overrightarrow{\texttt{mod}}, i, \overline{\texttt{id}})$ is represented by the parameters that are module-supplied by the $(i+1)st$ ini-module in $\overrightarrow{\texttt{mod}}$; such parameters must also belong to *(*)*, because no ini-module may have already permanently terminated them (they have just been supplied!); this means that *(b)* the parameters supplied by the $(i+1)st$ ini-module must be added to the parameter map.

In Algorithm 3, if the current ini-module is activated (if-branch, lines 4-18) then its input parameters are removed from the parameter map, while its output parameters are added to it. This satisfies the above requirements *(a)* and *(b)* because:

- the produced parameters of the current ini-module are added to the parameter map (since they are output parameters);
- the consumed parameters of the current ini-module are removed from the parameter map (since they are input parameters);
- the returning parameters of the current ini-module still stay in the parameter map (since they are input and output parameters, then they are removed and re-added with the new computed value).

If the current ini-module is not activated (else-branch, lines 19-32) then the recursive process is reactivated on the next ini-module, and the requirements *(a)* and *(b)* are equally satisfied. In fact, since the current ini-module is not activated, based on the two requirements, no parameter must be removed from the parameter map or added to it. □

After the execution of TryToActivate. Since the invariant has been proved, when the function CreateInstance exits the execution of recursive function TryToActivate, the invariant is true on the entire input. Indeed, at the end of the recursive process we have that:

$$(activated(\overrightarrow{\texttt{mod}}, size(\overrightarrow{\texttt{mod}}), \overline{\texttt{id}}) = \overrightarrow{\texttt{S}} \ \wedge \ RModules(\overrightarrow{\texttt{mod}}, size(\overrightarrow{\texttt{mod}})) \subseteq \overrightarrow{\texttt{S}}) \vee$$
$$activated(\overrightarrow{\texttt{mod}}, size(\overrightarrow{\texttt{mod}}), \overline{\texttt{id}}) = \blacksquare$$

which reduces to:

$$(activatedIniModules(\texttt{C}, \overline{\texttt{id}}) = \overrightarrow{\texttt{S}} \ \wedge \ RModules(\overrightarrow{\texttt{mod}}) \subseteq \overrightarrow{\texttt{S}}) \vee$$
$$activatedIniModules(\texttt{C}, \overline{\texttt{id}}) = \blacksquare$$

Moreover, the parameter map contains:

$$notPermTerminatedBy(\overrightarrow{\texttt{mod}}, size(\overrightarrow{\texttt{mod}}), suppliedBy(\overrightarrow{\texttt{mod}}, size(\overrightarrow{\texttt{mod}}), \overline{\texttt{id}}))$$
$$= notPermTerminatedBy(\overrightarrow{\texttt{mod}}, suppliedBy(\overrightarrow{\texttt{mod}}, \overline{\texttt{id}}))$$

Theorem 1 (Algorithm correctness)**.** Algorithm 1 is correct with respect to the ini-module activation semantics defined in Section 2.2, *i.e.,* given an object-creation expression

$$\texttt{E} = \texttt{new C}[\overline{\texttt{id := expr}}] \qquad (I)$$

one of the following two postconditions applies:

–O1 the ini-modules activated by the function CreateInstance are exactly those contained in $activatedIniModules(\texttt{C}, \overline{\texttt{id}})$, if $activatedIniModules(\texttt{C}, \overline{\texttt{id}}) \neq \blacksquare$; moreover, all the `required` ini-modules defined in the hierarchy of class `C` have been activated, and the parameter map is empty;

–O2 a runtime error is raised, if $activatedIniModules(\texttt{C}, \overline{\texttt{id}}) = \blacksquare$.

Moreover, the algorithm terminates on every possible input `E`.

Proof. We first prove the partial correctness of Algorithm 1 by showing that one and only one of the above postconditions *O1* or *O2* holds on all possible legal inputs I. Given an object-creation expression *exprNew*= $\texttt{new C}[\overline{\texttt{id := expr}}]$ and $\overrightarrow{\texttt{mod}} = \textsc{IniModulesInHierarchy}(\texttt{C})$, we introduce the following predicate describing a "proper" input:

$$\begin{aligned} ProperParams = \ & notPermTerminatedBy(\overrightarrow{\texttt{mod}}, suppliedBy(\overrightarrow{\texttt{mod}}, \overline{\texttt{id}})) = \emptyset \\ & \wedge \ RModules(\texttt{C}) \subseteq activatedIniModules(\texttt{C}, \overline{\texttt{id}}) \end{aligned}$$

meaning that the (directly and indirectly) supplied parameters in *exprNew* are all permanently terminated in *exprNew* and the `required` ini-modules defined through the hierarchy of `C` are all activated by *exprNew*.

We use this predicate to specify two contracts for our algorithm:

- First contract
  - Precondition (*I1*): *ProperParams*
  - Postcondition (*O1*): the activated ini-modules are $activatedIniModules(\mathtt{C}, \overline{\mathtt{id}}) = \overrightarrow{\mathtt{S}}$ with $\overrightarrow{\mathtt{S}} \neq \blacksquare \ \wedge RModules(\overrightarrow{\mathtt{mod}}) \subseteq \overrightarrow{\mathtt{S}} \ \wedge \ pm = \emptyset$
- Second contract
  - Precondition (*I2*): $\neg ProperParams$
  - Postcondition (*O2*): $activatedIniModules(\mathtt{C}, \overline{\mathtt{id}}) = \blacksquare$

Now, we want to prove that *I1* $\Rightarrow$ *O1* and that *I2* $\Rightarrow$ *O2*. We prove both by using the invariant we have proved to be correct.

First contract. Since by hypothesis we have that $RModules(\mathtt{C}) \subseteq activatedIniModules(\mathtt{C}, \overline{\mathtt{id}})$, at the end of TryToActivate we must have:

$$activatedIniModules(\mathtt{C}, \overline{\mathtt{id}}) = \overrightarrow{\mathtt{S}} \ \wedge \ RModules(\overrightarrow{\mathtt{mod}}) \subseteq \overrightarrow{\mathtt{S}}$$

with $\overrightarrow{\mathtt{S}} \neq \blacksquare$. Indeed, $RModules(\mathtt{C})$ can not be empty (we assumed that there is always at least one ini-module in `Object`), and a not-empty sequence can not be contained in $\blacksquare$.

Moreover, by hypothesis we have that the parameter map contains:

$$\begin{aligned} &= notPermTerminatedBy(\overrightarrow{\mathtt{mod}}, suppliedBy(\overrightarrow{\mathtt{mod}}, \overline{\mathtt{id}})) \\ &\overset{\text{hyp.}}{=} \emptyset \end{aligned}$$

Second contract. By hypothesis we have:

$$\begin{aligned} &notPermTerminatedBy(\overrightarrow{\mathtt{mod}}, suppliedBy(\overrightarrow{\mathtt{mod}}, \overline{\mathtt{id}})) \neq \emptyset \\ &\vee \ RModules(\mathtt{C}) \not\subseteq activatedIniModules(\mathtt{C}, \overline{\mathtt{id}}) \end{aligned}$$

As a consequence, one of the following must hold:

- if $notPermTerminatedBy(\overrightarrow{\mathtt{mod}}, suppliedBy(\overrightarrow{\mathtt{mod}}, \overline{\mathtt{id}})) \neq \emptyset$ then the parameter map is not empty and an error must have been raised (lines 13 and 26 of Algorithm 3);
- if $RModules(\mathtt{C}) \not\subseteq activatedIniModules(\mathtt{C}, \overline{\mathtt{id}})$ then also this implies that an error must have been raised (line 20 of Algorithm 3).

Termination. In Algorithm 3, the only two recursive calls of the procedure tryToActivate are those at lines 12 and 25. In both cases, the recursion terminates when $\overrightarrow{\mathtt{mod}}$ becomes empty (see the conditions respectively at lines 8 and 23). Before each recursive call the sequence $\overrightarrow{\mathtt{mod}}$ is diminished by an element (lines 9 and 24), ensuring that $\overrightarrow{\mathtt{mod}}$ will become empty in a finite number of steps, reaching one of the two end conditions. □

2.4 Note on I2 instructions

The definition of ini-module offers the possibility of including a `I2` instruction block after the `next_ini` call. Given an ini-module sequence $\overrightarrow{\texttt{mod}} = \texttt{I}_1, \ldots, \texttt{I}_n$, each ini-module $\texttt{I}_i$ in the sequence may define an `I2` instruction block, which will be executed *after* the ini-modules $\texttt{I}_{i+1}, \ldots, \texttt{I}_n$ in the sequence have been considered for activation. The examples of ini-modules we presented previously do not include any `I2` instruction block. However, sometimes it may be useful to define it. Let us consider, for example, an ini-module that opens a configuration file from which the values for initializing the fields of some object are read. In such a case, the presence of `I2` allows the class designer to define an ini-module whose responsibility is that of opening the file at the beginning of the initialization process, making it available to other ini-modules through the output parameter `configFile`, and closing it at the end:

```
required () initializes (configFile)
  File aConfigFile;
  aConfigFile := open(...);
  next_ini[configFile1 := aConfigFile];
  aConfigFile.close();
  // opens the config file, makes it available to other ini-modules, finally closes it

required (configFile1) initializes (configFile2)
  field1 := \\initialize the field by reading from the config file
  next_ini[configFile2 := configFile1];
  // initializes the first field

required (configFile2) initializes (configFile3)
  field2 := \\initialize the field by reading from the config file
  next_ini[configFile3 := configFile2];
  // initializes the second field

...

required (configFileN) initializes ()
  fieldN := \\initialize the field by reading from the config file
  next_ini[];
  // initializes the n-th field
```

Note that the parameters conveying the configuration file have been numbered from 1 to N to meet Constraint 1 (Ini-module signature uniqueness).

The `I2` instructions of the first ini-module, consisting in the closing of the file `configFile`, will be executed at the end of the activation of all the others ini-modules; indeed, in the procedure tryToActivate the execution of the instructions `I2` follows the recursive call (look at line 18 of Algorithm 3).

2.5 Discussion on the model

Kuśmierek *et al.* propose checks for ensuring that ini-modules in a hierarchy are well formed [4]. In Magda, these checks are conducted at compile time. In the following, we first discuss the main design choices that underpin our adaptation of ini-modules to the dynamically-typed context of Pharo. Then, we discuss in more detail how we tailor each of Magda's static checks to the dynamic nature of Pharo.

2.5.1 Design choices

How to verify checks. A first design choice concerns the approach to verifying checks on the initialization code. We can distinguish between two types of checks. *Correctness* checks represent constraints that are strict, *i.e.,* their violation must necessarily stop the execution of the program. *Quality* checks are not strict and verify some property of the code whose violation does not compromise the stability of the system. Quality checks, because of their non-stringent nature, can be also conducted through a sanity lint-like tool that raises warnings. On the contrary, correctness checks are more awkward, and they can be verified before the code is executed (via a static check) or while the code executes (a runtime error is raised in this case).

In Magda, the program is made non-executable in a preventive manner: a set of static checks makes the program non-compiling when one or more of the correctness checks are not verified. Such an approach is typical of a statically typed language and ensures a high degree of system stability. Kuśmierek proved the type-soundness of the Magda language [13]. As a consequence, if a Magda program compiles correctly, then no object-creation expression contained in the program can make it fail at runtime.

In contrast to static checking, dynamic checking may typically cause a program to fail at runtime. In such a context, correctness checks can be verified during the execution of the program. When a correctness check is violated, then the execution of the program is interrupted by raising a runtime error. For example, in Pharo the attempt to evaluate an object-creation expression whose supplied parameters do not activate one or more `required` ini-modules can be managed at runtime by raising an error, while in Magda this is checked statically before the execution of the program.

Ini-modules ordering. According to what we have seen in Section 2.2, the function *activatedIniModules* depends on the function *IModules* (Figure 2.3). In turn, the function *IModules* depends on the ini-modules $\overrightarrow{IM}$ defined into each class through the hierarchy. $\overrightarrow{IM}$ is supposed to be a sequence, which means that the ini-modules defined inside a class must be totally ordered. The reason underlying this design choice is that we want the behaviour of the initialization protocol to be deterministic. In other words, once the class designer has defined the ini-modules of a class, we want them to be always considered for activation in a fixed order, to avoid non-reproducible behaviours. For example, consider a pair of ini-modules, each of them initializing a same field `x`:

```
required (a) initializes ()
  x := a;
  next_ini[];

required (b) initializes ()
  x := b;
  next_ini[];
```

In such a situation, given an object-creation expression submitting both parameters `a` and `b`, considering first one ini-module or the other one yields to different behaviours because the field `x` will contain at the end the value `a` or `b`, depending on which ini-module is considered as the last one for activation.

In Magda, the order of the ini-modules is determined by their position in the source code. We recall that Magda's unit of reuse is the mixin, not the class, however with respect to the necessity of having an ordering this is irrelevant. An ini-module that is *textually below* other ini-modules is considered for activation first and those placed *textually above* afterwards. Moreover, in Magda a static check ensures that, in every mixin declared in a program, if some module outputs a parameter that is then taken in input by another module in the same mixin, then the latter is placed textually above the former. In Pharo, an approach on textual order was not possible because the programmer does not directly modify source files, but rather interacts with an IDE that allows the user to manipulate the object model. Such a diversity has been the starting point for reconsidering how to address the problem of ordering the ini-modules of a class. Indeed, we introduced a novel approach to ini-modules ordering, featuring the possibility for the programmer to abstract away both from order relationships that recur programmatically (*e.g.,* two ini-modules one producing the input parameters of the other one) and from order relationships that are non-influencing for computation (*e.g.,* two ini-modules for which the respective activation order is irrelevant). We will discuss in detail this novel approach to ini-modules ordering in Chapter 3.

2.5.2 Checks on well-formedness of ini-modules

We recall that the parameter map initially contains the new-supplied parameters, then it is extended with the output parameters of each activated ini-module, and decreased by the input parameters of the activated ini-modules (see Section 2.3). We also highlight that all the checks discussed in this section are correctness checks, and as such they are treated.

Check 1 (All output parameters must be assigned)**.** In Magda, all output parameters of an ini-module must be assigned in its body, in order to be ready to be consumed by other ini-modules. In Pharo, this check is conducted at runtime.

Check 2 (Output parameters must correspond to input parameters)**.** In Magda, each output parameter of an ini-module must be an input parameter of *exactly one* ini-module that will be considered after for activation. The fact that it must be *at least* input of one ini-module avoids the production of oversupplied parameters. The fact that it must be *at most* input of one ini-module is enforced by imposing that each *parameter name* is *unique*, with the aim of giving each parameter a non-ambiguous semantics.

In Pharo, we relaxed this constraint:

- Regarding the *at least* direction, we decided to postpone it verification from the creation of a new ini-module to the execution of an object-creation expression. This means that we tolerate defining an ini-module producing output parameters that are not taken in input by some ini-module to be considered after for activation; but we raise an error when such an ini-module is activated. Note that the activation algorithm fails when the parameter map is still not empty after trying to activate the last ini-module of the sequence (see lines 14-27 of Algorithm 3). An ini-module violating the *at least* direction of the constraint naturally leads to an error, when activated. Therefore its verification

is embedded in our algorithm. In the future, a quality check verifying preventively that a set of ini-modules is consistent on the basis of the *at least* direction of the constraint could be implemented as a sanity check.

- Regarding the *at most* direction, we discarded it because we believe that it is the developer's responsibility to decide if a certain parameter name must have more than one meaning, by appearing as an input parameter of more than one ini-module. However, we discussed in Section 2.2 that in such cases a parameter renaming preserving both ini-modules activation semantics and ordering is always possible. This means that using the same parameter name with more than one meaning should be, however, considered a bad programming practice. A quality check could be implemented in the future to verify if the *at most* direction of the constraint is satisfied.

Check 3 (Ini-module signature uniqueness)**.** Constraint 1 states that it must be forbidden to have two ini-modules with the same signature, *i.e.,* two ini-modules with same input and output parameters in the same class. In Magda, since each parameter name is unique, this constraint is automatically satisfied as a consequence. In Pharo, an error is raised at runtime if the class designer tries to define a new ini-module with the same signature of another one already defined in the same class.

Check 4 (On *fully applied* ini-modules)**.** In Magda, given an object-creation expression, an ini-module to be activated must have *all* its input parameters matched with a subset of the parameter map.

In Pharo this constraint is retained and checked at runtime: any ini-module with an input parameter not in the current parameter map is ignored, *i.e.,* not activated (see Definition 7). In both Magda and Pharo a non-fully-applied ini-module is *never* activated. The main difference is that in Magda this is checked statically, *i.e.,* the developer must remove this anomaly from the code, while in Pharo it is tolerated. From the point of view of a dynamically-typed language this is reasonable, given the inherently prototypical nature of such languages, as the developer might, sooner or later, add other choices of initialization that makes that ini-module fully applied and thus activated.

Check 5 (On *oversupplied* parameters)**.** In Magda, given an object-creation expression, only ini-modules *all* of whose output parameters are not in the parameter map can be activated. This prevents to have more than one way to calculate a certain output parameter, so to avoid ambiguity.

In Pharo this constraint is retained and checked at runtime: any ini-module that would add a produced parameter already present in the parameter map is ignored, *i.e.,* not activated (see Definition 7). In both Magda and Pharo an ini-module that outputs oversupplied parameters is *never* activated. The main difference is that in Magda the developer must remove this anomaly from the code, while in Pharo it is tolerated. In fact, in Pharo, the first-found ini-module computing an output parameter is activated, preventing the next ones computing the same output parameter to be activated (unless the parameter is consumed and re-added to the parameter map in between). This way the developer has the responsibility to put the

modules in the right order to get the desired effect. Regarding this design choice, a similar observation as for Check 4 applies here.

Check 6 (On *required* and *optional* modules)**.** In Magda, a static check ensures that all the ini-modules declared in mixins as `required` are activated by the supplied set of input parameters. Moreover, in FJMIP [5] each class must declare one and only one `required` ini-module, initializing all the fields of the class (this condition is verified by a static check). This ensures that, at the end of the initialization process, no field remains uninitialized.

In Pharo a tight property as the one guaranteed by FJMIP is not desirable, as lazy initialization (see Section 1.1.3) is a powerful and omnipresent initialization mechanism. However, the `required` ini-modules provide the class designer a mechanism to enforce a complete initialization of the state of the class, when needed. In Pharo, a runtime error is raised when a `required` ini-module is not activated.

In addition to those originally provided by Magda, in Pharo we also added the following check.

Check 7 (No field access in ini-modules bodies)**.** While in Magda it is possible to access field values in ini-modules bodies, in Pharo we decided to prohibit field access because it may introduce implicit dependencies between ini-modules, which can cause problems during software maintenance. In fact, the approach of ini-modules differs greatly from that of traditional constructors. A constructor is responsible for fully initializing a newly created instance of a class, while an ini-module represents only a piece of the whole initialization protocol. As a consequence, we think that how all these pieces combine together should be made as much as possible explicit, to facilitate code maintenance. Hence, we decided to force all the dependencies between ini-modules to be explicit in their signatures. However, this point would need a better investigation by experimenting with Pharo's code bases.

2.5.3 Class invariants

A *class invariant* is an assertion ϕ describing a property which must hold for all instances of a class. Class invariants must be established by the initialization and constantly maintained between calls to public methods. The class invariant constrains the state stored in the object.

As it is stated in [17], object creation may be seen as the operation that ensures that all instances of a class start their lives in a correct mode – one in which the invariant is satisfied.

However, in spite of its name, an invariant does not need to be satisfied at all times. At some intermediate stages, the invariant can not hold. Concerning object creation, the important thing is that the invariant is established *at the end* of the evaluation of an object-creation instruction. Let us explore more thoroughly what does this mean. Usually, an object-oriented language initializes the fields of an object based on their types (for example, the default value for a boolean field may be *false*). Meyer calls *creation procedure* each procedure of a class aimed at initializing newly created instances *after* the new instance has been created with all its fields initialized with their default values. Moreover, based on the *design by contract* [16], a creation procedure may require a precondition to ensure that a new

instance satisfying the invariant is successfully created. Putting together all the pieces, Meyer says that every creation procedure of a class `C`, when applied to arguments satisfying its precondition in a state where the fields have their default values, must yield a state satisfying ϕ.

Class invariants are inherited, that is, the invariants of all the ancestors of a class apply to the class itself. The class invariant for a class consists of any invariant assertions imposed locally on that class, logically "and-ed" with all the invariant clauses inherited from the class' ancestors. From the point of view of the initialization protocol, this means that a subclass may strengthen the invariant inherited from its ancestors, but it can not weaken it. In fact, for the Liskov substitution principle [14]:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Let us consider now ini-modules. When we say that ini-modules are modular, we mean that they differ from traditional constructors because each ini-module implements only a portion of the initialization of an object, whereas a constructor performs a fully initialization of an object. In other words, a traditional constructor is in a 1:1 relation with an object-creation expression, and it is a creation procedure in Meyer's sense. Ini-modules, instead, are in a 1:n relation with an object-creation expression, and the creation procedure in Meyer's sense coincides with the function CreateInstance which we presented in Section 2.3. As a consequence, with ini-modules we have only one creation procedure, the function CreateInstance, whose actual behaviour is determined by the ini-modules. Therefore, according to Meyer, we want to design the ini-modules for a class `C` in such a way that the function CreateInstance, when applied to arguments satisfying its precondition (*ProperParams* in a state where the fields have their default values, yields a state satisfying ϕ (see Section 2.3).

Consider an object-creation expression

`new C[id := expr]`

and let $\overline{\texttt{f}}$ be the set of the fields defined in `C`. Ini-modules allow the class designer to locally force a class invariant ϕ. In particular, if each field in $\overline{\texttt{f}}$ is initialized satisfying ϕ in at least one `required` ini-module, this guarantees that the initialization of all the fields defined in `C` satisfies ϕ for every object-creation expression.

We recall that our initialization algorithm retrieves the sequence of the ini-modules to be considered for activation through the function IniModulesInHierarchy. This function starts from the instantiated class `C` and goes up in the hierarchy, getting all the ini-modules defined through it and obtaining a totally ordered sequence $\overrightarrow{\texttt{mod}}$. This design choice is mainly motivated by the fact that, in this way, a subclass is able to introduce new manners of initializing a field inherited from a superclass, with no need of redefining the ini-module for setting the field.

Moreover, note that during the ini-module activation process (procedure TryToActivate) the class hierarchy is traversed two times:

- a first time in a *bottom-up* fashion: for each ini-module in $\overrightarrow{\texttt{mod}}$, the `I1` instructions are executed;
- a second time in a *top to bottom* fashion: the ini-modules in $\overrightarrow{\texttt{mod}}$ are traversed in a reversed order, and the `I2` instructions are executed. Indeed, any `I2` instruction block is evaluated only *after* the activation of the following ini-modules (see Section 2.4).

We recall that if each field in $\overline{\texttt{f}}$ is initialized satisfying ϕ in at least one `required` ini-module, this guarantees that ϕ is satisfied for every object-creation expression. If, in addition, the field assignments are all placed in the `I2` instructions of the ini-modules, then the field initialization will follow the hierarchical order, *i.e.,* from the root of the class hierarchy and going down to the class being instantiated. This is caused by how the ini-module activation process goes through the class hierarchy (see above). As a consequence, we believe that such a workaround allows a class designer to use the usual techniques to prove the correctness of a class with respect to the class invariant predicate.

As a final note, it is important to remark that here ini-modules are adapted to fit into a class-based style, whereas they were initially designed for a context of mixins in which inheritance is replaced by (or combined with) composition. As a consequence, we conjecture that alternative techniques for handling invariants should be studied. This is important also to study an alternative model of *stateful* traits with respect to the one of [2], basing on ini-modules.

Chapter 3

The ordering algorithm

In this section we discuss in detail how each class can specify a total ordering of its ini-modules, that is, how the function IniModulesOf (lines 3 and 6 of Algorithm 2) can be implemented. There are two options, which we expose in the next sections:

- the class designer directly specifies a total order of the ini-modules of a class;
- the class designer specifies a partial order of the ini-modules of a class, then the system infers a linear extension of it.

Both options require additional syntax to be added to the ini-modules.

3.1 Directly specifying a total ordering

The first option consists in forcing the class designer to explicitly establish a total ordering of the ini-modules of the class. For instance, each ini-module is assigned an order number:

optReq (`in_param_id`) `initializes` (`out_param_id`) `order` = `REAL`
`local_var`
`I1`
`next_ini`[`out_param_id` := `expr`];
`I2`

Figure 3.1 and 3.2 show a total ordering for the ini-modules of the classes `Rectangle2D` and `ColoredRectangle2D`.

Note that the order numbers are local to each class, because the order between ini-modules of different classes is determined upstream by the function IniModulesInHierarchy. In this case, the function IniModulesOf returns the ini-modules defined in a class simply by listing them according to the defined order numbers (Algorithm 4). Note also that we use rational numbers as order numbers, because a common need in software maintenance is inserting an ini-module between two other already present.

```
(angle, rad) initializes (coordX, coordY) order=1.0
  next_ini[coordX := cos(angle) * rad, coordY := sin(angle) * rad];
  // initializes lower-left corner from polar coordinates

(point) initializes (coordX, coordY) order=2.0
  next_ini[coordX := point.x, coordY := point.y];
  // initializes lower-left corner from a Point object

() initializes (coordX, coordY) order=3.0
  next_ini[coordX := 0, coordY := 0];
  // produces the default values for lower-left corner

(coordX, coordY) initializes () order=4.0
  x := coordX;
  y := coordY;
  next_ini[];
  // initializes lower-left corner from Cartesian coordinates

(aWidth) initializes () order=5.0
  width := aWidth;
  next_ini[];
  // initializes the width

(aHeight) initializes () order=6.0
  height := aHeight;
  next_ini[];
  // initializes the height
```

Figure 3.1: Specifying a total ordering for the ini-modules of class `Rectangle2D`.

```
(c, m, yc, k) initializes (red, green, blue) order=1.0
  next_ini[red   := 255 * (1 - c)  * (1 - k),
           green := 255 * (1 - m)  * (1 - k),
           blue  := 255 * (1 - yc) * (1 - k)];
  // initializes color from a CMYK palette

(red, green, blue) initializes () order=2.0
  r := red;
  g := green;
  b := blue;
  next_ini[];
  // initializes color from a RGB palette
```

Figure 3.2: Specifying a total ordering for the ini-modules of class `ColoredRectangle2D`.

Algorithm 4 Retrieves the ini-modules locally defined in a class `C`

Input: A class `C`.
Output: An ordered sequence of the ini-modules locally defined in `C`.

```
1: function IniModulesOf(C)
2:     mod⟶ ⟵ a sequence of ini-modules locally defined in C
3:     sort mod⟶ by order numbers
4:     return mod⟶
5: end function
```

The total ordering can be specified also in some other comparable way. For example, in Magda the ini-modules of a mixin are totally ordered based on their textual ordering in the source code: the one which is *textually below* is executed first and those placed *textually above* afterwards [13].

3.2 Specifying a partial ordering and inferring a linear extension

Specifying a total ordering of the ini-modules of a class may result in an unnecessary and annoying task for two reasons:

1. Some order relations between ini-modules recur programmatically. Look for example at Figure 3.1: the ini-modules `(angle, rad)(coordX, coordY)`, `(point)(coordX, coordY)` and `()(coordX, coordY)` must all precede the ini-module `(coordX, coordY)()`. In the same way, in Figure 3.2 the ini-module `(c, m, yc, k)(red, green, blue)` must precede ini-module `(red, green, blue)()`. This is because an ini-module outputting parameters that are input parameters of another ini-module should usually precede it. Another recurring order relation is that between ini-module `()(coordX, coordY)` and ini-modules `(angle, rad)(coordX, coordY)` and `(point)(coordX, coordY)`, which must both precede it. This is because ini-modules producing default values (usually with an empty set of input parameters) should follow in the given order other ini-modules producing the same output parameters.

2. For some pairs of ini-modules, their relative ordering is irrelevant. Consider for example ini-modules `(aWidth)()` and `(aHeight)()`: initializing first the `width` and then the `height` or vice versa produce the same result.

To take in account the previous two points, we introduce two kinds of order constraints. Each order constraint states that an ini-module must be considered for activation before or after another one.

Default order constraints. We define a pair of default order constraints between two ini-modules A and B that apply by default when the signatures of the ini-modules exhibit particular patterns. This avoids the class designer from having to specify order relations that

recur programmatically between ini-modules. In particular, we introduce two rules, Rule 1 having priority over Rule 2 (*i.e.,* Rule 2 is tried only when Rule 1 does not apply):

- Rule 1: if A outputs at least one parameter that is consumed by B then A is tried first;
- Rule 2: if A takes as input parameters a strict subset of B's input parameters, and A and B share at least one output parameter, then B is tried first.

Rule 1 captures the order relation between two ini-modules, the second ini-module consuming at least one parameter that is outputted by the first one. Rule 2 establishes that, when two ini-modules share at least one output parameter, the one requiring more data in input must be tried first as it is probably more precise and increases the chance that all parameters will be terminated before the end of the activation process.

In the rectangle example shown in Figures 2.5 and 2.6, the following default constraints apply:

- according to Rule 1, ini-modules `Rectangle2D»(angle, rad)(coordX, coordY)`, `Rectangle2D»(point)(coordX, coordY)` and `Rectangle2D»()(coordX, coordY)` must be considered for activation before ini-module `Rectangle2D»(coordX, coordY)()`;
- according to Rule 2, ini-modules `Rectangle2D»(angle, rad)(coordX, coordY)` and `Rectangle2D»(point)(coordX, coordY)` must be considered for activation before ini-module `Rectangle2D»()(coordX, coordY)`;
- according to Rule 1, ini-module `ColoredRectangle2D»(c, m, yc, k)(red, green, blue)` must be considered for activation before ini-module `ColoredRectangle2D»(red, green, blue)()`.

Note that a possible application of Rule 2 is for ini-modules producing default values, such as `Rectangle2D»()(coordX, coordY)`. Typically, such ini-modules have no input parameters. Therefore, Rule 2 imposes that these ini-modules are considered for activation only *after* other ini-modules producing the same parameters starting from one or more input parameters.

Explicit order constraints. We also make it possible for the class designer to declare explicit order constraints. Given two ini-modules A and B, it is possible to declare that A *before* B or that A *after* B. We use the signature of ini-modules to refer to them unequivocally:

(in_param_id) (out_param_id) before (in_param_id) (out_param_id)
(in_param_id) (out_param_id) after (in_param_id) (out_param_id)

Note that the default constraints are deduced from the declarations of ini-modules, based on their signatures. Instead, the explicit constraints are declared by the class designer. The idea behind is that default constraints capture most of the dependencies between ini-modules. Those dependencies induce a partial order on ini-modules. Explicit constraints are to be used to impose other order constraints that are not captured by the default ones. Introducing

names for ini-modules would simplify the definition of explicit constraints, resulting in a less verbose, and thus more readable, code. However, names for ini-modules are not necessary for their activation since ini-modules are activated by the parameters supplied in an object-creation expression. We could consider named ini-modules if we introduce a form of overriding between ini-modules. For the moment, our Pharo prototype partially reduces the verbosity of the code, because ini-modules are implemented as objects that can be assigned to variables, and these variables can be used to refer ini-modules while defining explicit ordering constraints (see Section 4.1.2).

The rectangle example does not need any explicit constraint to be specified. This means that it can simply be written as it was initially defined in Figures 2.5 and 2.6. Indeed, all the needed order constraints are deduced by default from the ini-module declarations, with no need of explicit constraints.

Another example. We explain the use of default and explicit constraints through another example. Consider a class `Person`, with three fields, two for `name` and `surname`, and one for a `nickname`. All people have a name and a surname, therefore the values for both of them must be submitted in every object-creation expression. The nickname is optional; if no nickname is submitted in an object-creation expression, then a default one is automatically generated from the name and the surname. Every person is also associated to a code, and this code is calculated from name and surname, or from the nickname: if surname is provided, this takes precedence to calculate the code.

Figure 3.3 shows a set of ini-modules for the class `Person`:

- `(aNickname)(aNickname, aCode)` (line 2) creates a code from a nickname; note that the parameter `aNickname` is returning as it is used for computation but not consumed by the ini-module;
- `(aName, aSurname)(aName, aSurname, aCode)` (line 8) creates a code starting from name and surname; note that the parameters `aName` and `aSurname` are returning as they are used for computation but not consumed by the ini-module;
- since we want the code to be calculated from the nickname and not from name/surname, if both are submitted in an object-creation expression, we need to explicitly state that the ini-module `(aNickname)(aNickname, aCode)` must be considered *before* the ini-module `(aName, aSurname)(aName, aSurname, aCode)` (line 9);
- `(aName, aSurname)(aName, aSurname, aNickname)` (line 16) creates a default nickname starting from name and surname; note that the parameters `aName` and `aSurname` are returning as they are used for computation but not consumed by the ini-module;
- each of the last three ini-modules consumes one or more parameters by setting the respective fields.

Concerning default order constraints, in this example only Rule 1 applies, in the following cases:

```
1  optional (aNickname) initializes (aNickname, aCode) //I1
2    //...creates a code starting from a nickname...
3    next_ini[aNickname := aNickname, aCode := //the created code];
4
5  optional (aName, aSurname) initializes (aName, aSurname, aCode) //I2
6    //...creates a code starting from name and surname...
7    next_ini[aName := aName, aSurname := aSurname, aCode := //the created code];
8
9  (aNickname)(aNickname, aCode) before (aName, aSurname)(aName, aSurname, aCode);
10
11 optional (aName, aSurname) initializes (aName, aSurname, aNickname) //I3
12   //...creates a default nickname from name and surname...
13   next_ini[aName := aName, aSurname := aSurname, aNickname := //the created nickname];
14
15 required (aName, aSurname) initializes () //I4
16   name := aName;
17   surname := aSurname;
18   next_ini[];
19 //sets name and surname
20
21 required (aNickname) initializes () //I5
22   nickname := aNickname;
23   next_ini[];
24 //sets the nickname
25
26 required (aCode) initializes () //I6
27   code := aCode;
28   next_ini[];
29 //sets the code
```

Figure 3.3: The ini-modules for the class `Person`.

$$\frac{A \textit{ before } B}{A \lhd B} \lfloor \textsc{BeforeConstr} \rfloor$$

$$\frac{A \textit{ after } B}{B \lhd A} \lfloor \textsc{AfterConstr} \rfloor$$

Figure 3.4: Definition of explicit constraints.

$$\frac{A \text{ Rule 1 } B}{A \lessdot B} \lfloor \textsc{Rule1Appl} \rfloor$$

$$\frac{A \text{ Rule 2 } B}{A \lessdot B} \lfloor \textsc{Rule2Appl} \rfloor$$

Figure 3.5: Definition of default constraints.

- ini-module `(aNickname)(aNickname, aCode)` must be considered for activation before ini-modules `(aNickname)()` and `(aCode)()`;
- ini-module `(aName, aSurname)(aName, aSurname, aCode)` must be considered for activation before ini-modules `(aName, aSurname)()` and `(aCode)()`;
- ini-module `(aName, aSurname)(aName, aSurname, aNickname)` must be considered for activation before ini-modules `(aName, aSurname)()` and `(aNickname)()`.

Based on these two kinds of order constraints, default and explicit, we want now to formally define a partial order with the following requisite: we want explicit constraints to always have priority over default constraints, *i.e.,* we want default constraints to apply only in the absence of explicit constraints that contradict them.

3.2.1 Formal definition of the partial order $\leq$

Given a set I of the ini-modules of a class C, we associate this set of ini-modules with a set $\mathbb{C} =< \mathbb{E} \cup \mathbb{D} >$. This set is the union of two sets of order constraints, each of them specifying an order between two ini-modules A and B. These constraints can be of two different types:

- We write $A \lhd B$ if A *before* B or B *after* A does hold (see Figure 3.4). We call $\mathbb{E}$ the set of all the constraints of this type, *i.e.,* the set of all the explicit constraints.
- We write $A \lessdot B$ if Rule 1 or 2 applies (see Figure 3.5). We call $\mathbb{D}$ the set of all the constraints of this type, *i.e.,* the set of all the default constraints.

Example. In the example of the class `Person`, we have the only explicit constraint:

$$(\texttt{aNickname})(\texttt{aNickname}, \texttt{aCode}) \lhd (\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aCode})$$

while the default constraints are formalized as shown below:

- ini-module `(aNickname)(aNickname, aCode)` must be considered for activation before ini-modules `(aNickname)()` and `(aCode)()` (rule ⌊RULE1APPL⌋);

$$(\texttt{aNickname})(\texttt{aNickname}, \texttt{aCode}) \lessdot (\texttt{aNickname})()$$
$$(\texttt{aNickname})(\texttt{aNickname}, \texttt{aCode}) \lessdot (\texttt{aCode})()$$

- ini-module `(aName, aSurname)(aName, aSurname, aCode)` must be considered for activation before ini-modules `(aName, aSurname)()` and `(aCode)()` (rule ⌊RULE1APPL⌋);

$$(\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aCode}) \lessdot (\texttt{aName}, \texttt{aSurname})()$$
$$(\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aCode}) \lessdot (\texttt{aCode})()$$

- ini-module `(aName, aSurname)(aName, aSurname, aNickname)` must be considered for activation before ini-modules `(aName, aSurname)()` and `(aNickname)()` (rule ⌊RULE1APPL⌋).

$$(\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aNickname}) \lessdot (\texttt{aName}, \texttt{aSurname})()$$
$$(\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aNickname}) \lessdot (\texttt{aNickname})()$$

We want now to define more precisely how default and explicit constraints combine together to determine the reciprocal order of two ini-modules A and B. In particular, we recall that we want the explicit constraints to always take precedence over the default ones.

Definition 15 (Direct order constraint)**.** We say that an ini-module A must be considered for activation before an ini-module B (written $A \prec B$ and called *direct constraint*), if at least one of the following holds:

- an explicit order between A and B is imposed;
- a default order between A and B that is not contradicted by some explicit order is inferred.

The rules for calculating the relation $\prec$ are shown in Figure 3.6. We call $\mathbb{C}'$ the set of all the direct constraints $A \prec B$.

The inference rule ⌊EXPCONSTR⌋ states that, if any explicit constraint $A \lhd B$ exists, then we can immediately conclude that $A \prec B$. This is because we want each explicit order constraint to be always satisfied. The inference rule ⌊DEFCONSTR⌋ determines when a default constraint $A \lessdot B$ applies, being promoted to a direct constraint. The idea behind this rule is that a default order constraint should be valid unless the class designer states an explicit constraint that contradicts it.

Example. The direct order constraints for the class `Person` are shown in Figure 3.7. The only explicit constraint is promoted to direct constraint, based on rule ⌊EXPCONSTR⌋; all the

$$\frac{A \lhd B}{A \prec B} \lfloor \textsc{ExpConstr} \rfloor$$

$$\frac{A \lessdot B \qquad \neg(B \lhd A)}{A \prec B} \lfloor \textsc{DefConstr} \rfloor$$

Figure 3.6: Definition of direct order constraints.

$$\begin{gathered}\mathbb{C}' = \{(\texttt{aNickname})(\texttt{aNickname}, \texttt{aCode}) \prec (\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aCode}),\\ (\texttt{aNickname})(\texttt{aNickname}, \texttt{aCode}) \prec (\texttt{aNickname})(),\\ (\texttt{aNickname})(\texttt{aNickname}, \texttt{aCode}) \prec (\texttt{aCode})(),\\ (\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aCode}) \prec (\texttt{aName}, \texttt{aSurname})(),\\ (\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aCode}) \prec (\texttt{aCode})(),\\ (\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aNickname}) \prec (\texttt{aName}, \texttt{aSurname})(),\\ (\texttt{aName}, \texttt{aSurname})(\texttt{aName}, \texttt{aSurname}, \texttt{aNickname}) \prec (\texttt{aNickname})()\}\end{gathered}$$

Figure 3.7: Direct order constraints for the class `Person`.

default constraints are promoted as well to direct constraints, based on rule $\lfloor \textsc{DefConstr} \rfloor$, because the only explicit constraint does not contradict any of them.

With the aim of ordering ini-modules basing on direct order constraints, we introduce now a relation $\leq$ on the set I of ini-modules and then we prove that, under some assumptions, it represents a partial order.

Definition 16 (Partial order $\leq$)**.** We define the binary relation $\leq$ over the set of ini-modules I as shown in Figure 3.8.

$$\frac{A = B}{A \leq B} \lfloor \textsc{Reflexivity} \rfloor$$

$$\frac{A \prec B}{A \leq B} \lfloor \textsc{DirectConstr} \rfloor$$

$$\frac{A \prec I_1, I_1 \prec I_2, ..., I_n \prec B \ \text{ with } \ n \geq 1}{A \leq B} \lfloor \textsc{Transitivity} \rfloor$$

Figure 3.8: Definition of the partial order $\leq$.

Rules are very simple and self-explanatory. We do not show extensively the partial order for the class `Person` because it is fairly obvious (for example, for transitivity we have that $(\texttt{aNickname})(\texttt{aNickname}, \texttt{aCode}) \leq (\texttt{aName}, \texttt{aSurname})()$).

Theorem 2. If the set of direct constraints $\mathbb{C}'$ is acyclic then the binary relation $\leq$ over the set of ini-modules I, as defined by Definition 16, is a partial order.

Proof. We must prove that the relation $\leq$ is:

- reflexive;
- antisymmetric;
- transitive.

Reflexivity. We must prove that $A \leq A \quad \forall A \in I$. This trivially follows from rule $\lfloor$Reflexivity$\rfloor$ of the definition.

Antisymmetry. We must prove that $A \leq B \quad \wedge \quad B \leq A \quad \Rightarrow \quad A = B \quad \forall A, B \in I$.

The relation $\leq$ is defined by three different rules, but these rules have defined priorities (they are not in an exclusive or). This means that, if $A \leq B$ results from one of the rules, it is not possible to obtain also the opposite result $B \leq A$ by applying another rule. Therefore, the antecedent of our thesis can never be true in such a way. Consequently, we just have to prove the thesis for each single rule:

- For rule $\lfloor$Reflexivity$\rfloor$, the thesis is trivially proved.
- Concerning $\lfloor$DirectConstr$\rfloor$, the antecedent of our thesis would be true if $A \prec B$ and $B \prec A$. But this would mean having a cycle in the set $\mathbb{C}'$.
- Concerning rule $\lfloor$Transitivity$\rfloor$, for the antecedent of our thesis to be true, two constraint chains of the following form should exist:

$$A \prec I_1, I_1 \prec I_2, \ldots, I_n \prec B \quad \text{with} \quad n \leq 1$$

$$B \prec K_1, K_1 \prec K_2, \ldots, K_m \prec A \quad \text{with} \quad m \leq 1$$

But this would mean that the set of constraints $\mathbb{C}'$ would contain a cycle.

Transitivity. We must prove that $A \leq B \quad \wedge \quad B \leq C \quad \Rightarrow \quad A \leq C \quad \forall A, B, C \in I$.

We have to consider all the ways in which the antecedent of the thesis can be realised:

- If both the order relationships of the antecedent are inferred by the rule $\lfloor$Reflexivity$\rfloor$, the thesis is trivially proved: we have $A = B$ and $B = C$, consequently $A = C$ and therefore $A \leq C$ based on the same rule.

- If one relationship is obtained by the rule $\lfloor$REFLEXIVITY$\rfloor$ and the other one by one of the other two, the thesis is equally verified. If we have $A = B$ based on $\lfloor$REFLEXIVITY$\rfloor$ and $B \leq C$ by another rule, then it follows $A \leq C$. Obviously, the same if $B = C$ based on $\lfloor$REFLEXIVITY$\rfloor$ and $A \leq B$ by another rule, $A \leq C$ follows.

- If both the order relationships of the antecedent are inferred by the rule $\lfloor$DIRECTCONSTR$\rfloor$, we have that $A \prec B$ and $B \prec C$. Therefore, based on rule $\lfloor$TRANSITIVITY$\rfloor$, we can infer $A \leq C$.

- If the first relationship is obtained by the rule $\lfloor$DIRECTCONSTR$\rfloor$ and the second one by the rule $\lfloor$TRANSITIVITY$\rfloor$, then we have $A \prec B$ and a chain $B \prec I_1 \prec \cdots \prec I_n \prec C$, with $n \geq 1$. Therefore we can infer $A \leq C$ by rule $\lfloor$TRANSITIVITY$\rfloor$. The converse is analogous.

- If both the order relationships of the antecedent are inferred by the rule $\lfloor$TRANSITIVITY$\rfloor$, then we have two chains $A \prec I_1 \prec \cdots \prec I_n \prec B$ and $B \prec K_1 \prec \cdots \prec K_m \prec C$, with $n \geq 1$ and $m \geq 1$. Therefore, we can infer $A \leq C$ by rule $\lfloor$TRANSITIVITY$\rfloor$.

□

3.2.2 Calculating a linear extension of the partial order $\leq$

By exploiting default and explicit constraints, the class designer can specify how the ini-modules of a class must be considered for activation. We have seen that such a specification allows the class designer to handle only the needed order constraints between ini-modules, abstracting away from orderings that are irrelevant. However, the function INIMODULESOF needs to produce, for each class it is applied to, a *totally* ordered sequence of the ini-modules defined in that class. This is because:

- we need to obtain a sequence of the ini-modules defined in a class hierarchy to be used for the activation process (line 5 of Algorithm 1);

- we want this sequence to be the same at each execution to ensure that our algorithm is deterministic.

In Section 3.1 we have seen a first version of the function INIMODULESOF, supposing a total order was directly specified by the class designer via order numbers. In this section we show, given a class C and a partial order $\leq$ on its set of ini-modules I – specified through default and explicit constraints – how we can define a function INIMODULESOF that computes a total order that is compatible with the imposed partial order. In order theory, such a total order is called a *linear extension* of a partial order.

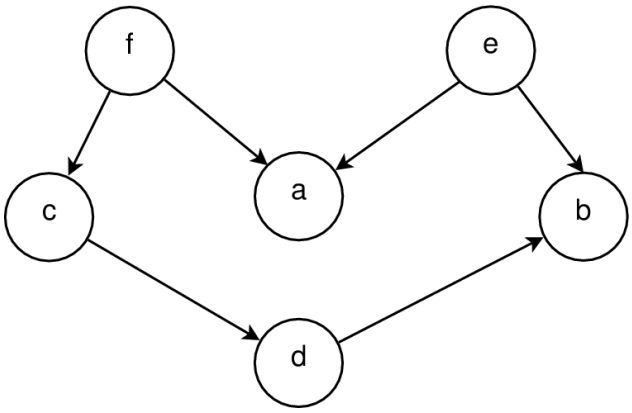

Figure 3.9: An example of a DAG.

Topological sorting

We introduce some useful definitions.

Definition 17 (Linear extension of $\leq$)**.** Given any partial orders $\leq$ and $\leq^*$ on a set X, $\leq^*$ is a *linear extension* of $\leq$ exactly when (1) $\leq^*$ is a total order, *and* (2) for every x and y in X, if $x \leq y$, then $x \leq^* y$.

For example, consider a set $A = \{a, b, c, d\}$ and a partial order on it defined as $\leq = \{(a, b), (c, d)\}$. The linear extensions of $\leq$ are "a, b, c, d", "a, c, b, d", "a, c, d, b", "c, a, b, d", "c, a, d, b", and "c, d, a, b", all of which have a before b and c before d. Note that here and in the following, for brevity we write total orderings as sequences.

A linear extension of a partial order can be computed in linear time by using an algorithm for constructing a *topological sorting* of a directed acyclic graph (DAG):

Definition 18 (Topological sorting)**.** Given a directed acyclic graph (DAG), a *topological sorting* for it is a total ordering of its nodes such that for every directed edge uv from node u to node v, u comes before v in the ordering.

For example, given the DAG shown in Figure 3.9, a topological sorting of it is "f, e, c, d, b, a". There can be more than one topological sorting for a graph. For example, another topological sorting of the graph in Figure 3.9 is "e, f, c, d, b, a".

A topological sorting is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG).

Any DAG has at least one topological sorting, and algorithms are known for constructing a topological sorting of any DAG in linear time. Moreover, they typically allow the user to detect the presence of cycles in the graph.

Below we present a pair of well-known algorithms for topological sorting, which have running time linear in the number of nodes plus the number of edges, *i.e.,* $O(|V| + |E|)$.

Kahn's algorithm. A first algorithm for topological sorting was described by Kahn [12] and is shown in Algorithm 5. It works by incrementally choosing nodes in the same order as the

Algorithm 5 Kahn's algorithm

Input: A graph $G = (V, E)$.
Output: A topological sort of G.

```
1: function TopologicalSort(G)
2:     L ⟵ empty list that will contain the sorted elements
3:     S ⟵ set of all nodes with no incoming edges
4:     while S is non-empty do
5:         remove a node n from S
6:         add n to tail of L
7:         for all node m with an edge e from n to m do
8:             remove edge e from the graph
9:             if m has no other incoming edges then insert m into S
10:            end if
11:        end for
12:    end while
13:    if graph has edges then return error (graph has at least one cycle)
14:    else return L (a topologically sorted order)
15:    end if
16: end function
```

eventual topological sort and by inserting them into a list L (line 2). First, a list of "starting nodes" which have no incoming edges is found and inserted into a set S (line 3); if the graph is acyclic, at least one of such nodes must exist. At each iteration step, a node is extracted from S and put in the list L (lines 5-6). Then, the edges outgoing from the current node are removed from the graph (line 8), and new nodes that remain with no incoming edges are added to S (line 9). When S is empty, if the graph still has incoming edges, this means that it is not a DAG (it contains at least one cycle), therefore a topological sorting is impossible and an error is raised (line 13); otherwise, the graph is a DAG and a solution is contained in the list L (line 14).

As said previously, the solution is not necessarily unique: the structure S is a set, therefore at this level of abstraction the extraction of an element is random.

Depth-first search. Another approach to topological sorting is to appropriately modify the algorithm for depth first traversal (DFS) of a graph. In DFS, we start from a node, we first visit it and then recursively call DFS for its adjacent nodes, exploring as far as possible along each branch before backtracking. For example, a DFS of the graph shown in Figure 3.9 is "f, c, d, b, a, e", but it is not a topological sorting. Indeed, in a topological sorting, unlike DFS, the node 'e' should come before the node 'a'. Therefore, topological sorting is different from DFS. To modify DFS to find topological sorting of a graph, we do not add the node immediately to the resulting sequence, we first recursively call topological sorting for all its adjacent nodes, and only at the end we add it as the head of the sequence. In such a way, a node is added to the resulting sequence only when all of its adjacent nodes (and their adjacent nodes and so on) are already in the sequence. The depth-first-search-based algorithm shown

Algorithm 6 Depth-first search

Input: A graph $G = (V, E)$.
Output: A topological sort of G.

```
1: function TopologicalSort(G)
2:     L ⟵ empty list that will contain the sorted nodes
3:     while there are unmarked nodes do
4:         select an unmarked node n
5:         visit(n)
6:     end while
7: end function
8: procedure visit(node n)
9:     if n has a temporary mark then
10:        stop (not a DAG)
11:    end if
12:    if n is not marked (i.e. has not been visited yet) then
13:        mark n temporarily
14:        for all node m with an edge from n to m do visit(m)
15:        end for
16:    end if
17:    mark n permanently
18:    unmark n temporarily
19:    add n to head of L
20: end procedure
```

in Algorithm 6 is the one described by [8] and it has been first described by [20].

Using topological sorting to calculate a totally ordered set of ini-modules

The goal of this section is to compute a linear extension $\leq^*$ of the partial order $\leq$ we defined in Section 3.2.1.

We formulate our goal as the problem of finding a topological sorting for a graph constructed as follows.

Definition 19 (Ini-modules graph)**.** Given the set I of the ini-modules of a class C, we define a graph $G = (I, \mathbb{C}')$ whose nodes are the ini-modules in I and whose edges are the direct constraints in $\mathbb{C}'$, *i.e.,* there is an edge from a node `I1` to another node `I2` if and only if `I1` $\prec$ `I2`. We refer to this graph as the *ini-modules graph* for the class C.

Figure 3.10 shows the ini-modules graph for the class `Person`, in which we have called the ini-modules with the labels we assigned to them in Figure 3.3. The edges have been constructed based on the set $\mathbb{C}'$ shown in Figure 3.7.

Note that the shown ini-module graph is a DAG. Indeed, it contains no directed cycles. For an ini-module graph, to be a DAG means that the set of direct constraints $\mathbb{C}'$ is acyclic,

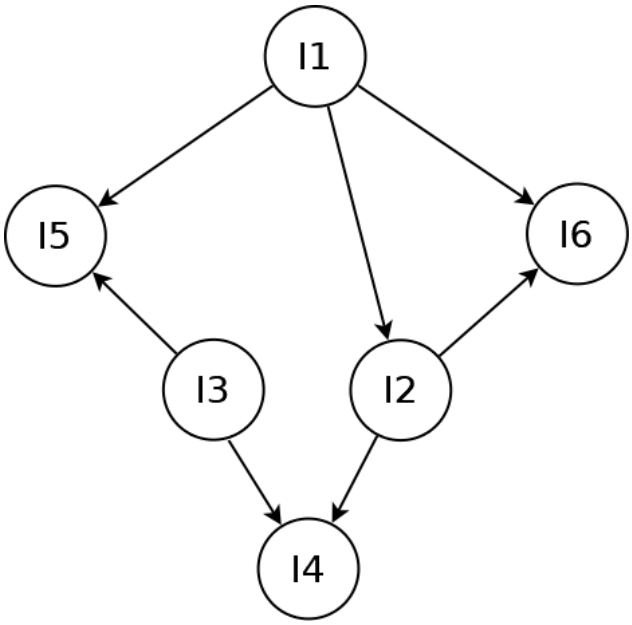

Figure 3.10: The ini-modules graph for the class `Person`.

that is an hypothesis of Theorem 2.

To obtain a linear extension $\leq^*$ of the partial order $\leq$, we can then apply one of the algorithms shown in the previous section. Algorithm 7 shows the function IniModulesOf that returns a sorted sequence of the ini-modules defined in a class. An example of total ordering for the ini-modules of the class `Person`, which can be obtained for example by applying Kahn's algorithm, is "`I1`, `I3`, `I2`, `I5`, `I6`, `I4`".

Cycle detection. If the ini-modules graph is not a DAG, *i.e.,* the set of direct constraints $\mathbb{C}'$ is cyclic, then the topological algorithm will detect the presence of one or more cycles, and an error will be raised (lines 7-9 of Algorithm 7). Note that such an error will be typically raised when adding the direct constraint that creates the cycle, *i.e.,* during the definition of an ini-module or the adding of an explicit constraint. In this case, the binary relation $\leq$ over the set of ini-modules I is not a partial order and a linear extension $\leq^*$ of it can not be computed.

Note that a cycle in the ini-modules graph can arise in different situations:

- from a set of default constraints: consider for example two ini-modules A and B such that A outputs an input parameter of B, and vice versa;
- from a set of explicit constraints: consider for example two ini-modules A and B, and two explicit constraints A *before* B and B *before* A;
- from a combination of default/explicit constraints: consider for example three ini-modules A, B and C such that A outputs an input parameter of B, B outputs an input parameter of C, plus an explicit constraint C *before* A.

Algorithm 7 Retrieves the ini-modules locally defined in a class `C`

Input: A class `C`.
Output: An ordered sequence of the ini-modules locally defined in `C`.

```
1: function IniModulesOf(C)
2:     mod⃗ ⟵ a sequence of ini-modules locally defined in C
3:     ec ⟵ compute explicit constraints based on rules in Figure 3.4
4:     dc ⟵ compute default constraints based on rules in Figure 3.5
5:     ℂ′ ⟵ compute direct constraints based on rules in Figure 3.6, starting from ec, dc
6:     build the ini-modules graph G starting from mod⃗ and ℂ′
7:     if G is not a DAG then
8:         raise an error
9:     end if
10:    sort mod⃗ by calculating a (deterministic) topological sort of G
11:    return mod⃗
12: end function
```

Forcing the determinism. A crucial point is how to force the determinism of the total ordering, that is, we want the function IniModulesOf to be deterministic. However, we have seen that, given a DAG, the topological sorting is in general not unique. Both Khan's algorithm and depth-first search do not guarantee that, given a DAG, the obtained total ordering is always the same on different executions. For example, at line 5 Kahn's algorithm removes a node n from the set S: which node is extracted from S at each step is not fixed, thus the result can change across executions. To make the algorithm deterministic, we must establish a policy by which to choose a node at each step.

Indeed, given an object-creation expression `new C[id := expr]`, its semantics (*i.e.,* the sequence of the activated ini-modules) is given by the execution of *activatedIniModules*(`C`, $\overline{\texttt{id}}$), which depends in turn on the sequence of ini-modules considered for activation (function *IModules*, look at Figure 2.3). We need to use some deterministic criterion for selecting the node to be considered at each step. Such a criterion may be based on a hash function that is:

- *deterministic*, *i.e.,* given a node it always returns the same hash value, on different executions and machines;
- *perfect*, *i.e.,* it is an injective function.

Note that this design choice guarantees the obtained total ordering to be deterministic, but not *predictable*. This can be problematic in situations like the following:

```
optional (a) initializes (b)
  next_ini[b := ...];

optional (b) initializes (c)
  next_ini[c := ...];

optional (b) initializes (d)
  next_ini[d := ...];
```

```
required (c) initializes ()
  fieldC := c;
  next_ini[];

required (d) initializes ()
  fieldD := d;
  next_ini[];
```

because the behaviour depends on the ordering in which the ini-modules `(b)(c)` and `(b)(d)` are considered for activation: given an object-creation expression supplying parameter `a`, if the ini-module `(b)(c)` is considered first then the `fieldC` field will be set; otherwise, if the ini-module `(b)(d)` is considered first then the `fieldD` field will be set. However, as we will see in the next chapter, the GUI of our Pharo prototype displays the total ordering generated by the system, and in such situations the class designer has a total control over the total ordering via the explicit constraints. Indeed, through the explicit constraints it is possible to force any desired ordering, potentially also a total ordering of the defined ini-modules.

Chapter 4

Pharo implementation

This chapter introduces ini-modules in Pharo and is organized as follows:

- In Section 4.1 we present our Pharo implementation of the ini-modules model we presented in Chapter 2. This implementation features ini-modules implemented as objects and allows the class designer to specify a partial ordering of the ini-modules defined in a class, through default and explicit constraints, as we saw in Section 3.2.
- Section 4.2 briefly outlines an alternative implementation that we tried before getting to the current one. In particular, we highlight its drawbacks that led us to move to the current implementation.

4.1 Ini-modules in Pharo

We show how to initialize the instances of the class `Rectangle2D` in Pharo, by using ini-modules:

```
Object subclass: #Rectangle2D
  instanceVariableNames: 'x y width height'
  classVariableNames: ''
  category: 'IniModules-Tests-Classes'
```

As usual, we want the lower-left corner to be initialized in three different ways: (1) by providing Cartesian coordinates directly, (2) by providing polar coordinates, and (3) from a `Point` object.

In standard Pharo, this would mean implementing instance-creation methods on class side and mutators on instance side. If for some reason the class designer wants the instances of `Rectangle2D` to be immutable, things become even more complex because mutators are mandatory.

With ini-modules, we can obtain the desired behavior as shown in Figure 4.1. Each ini-module is defined via a block, which is passed as an argument to one of two different messages, `addIniModule:` or `addRequiredIniModule:`. The difference between the two messages is just that the created ini-module is marked as optional or required, respectively.

```
"Rectangle2D>>(x y)()
initialize lower-left corner from Cartesian coordinates"
Rectangle2D addRequiredIniModule: [ :x :y :instVars | instVars at: #x put: x;
                                                        at: #y put: y.
                                                    IniModule nextIni: {  }. ].

"Rectangle2D>>(angle rad)(x y)
initialize lower-left corner from polar coordinates"
Rectangle2D addIniModule: [ :angle :rad | IniModule nextIni: {#x -> (angle cos * rad).
                                                              #y -> (angle sin * rad)} ].

"Rectangle2D>>(point)(x y)
initialize lower-left corner from a Point object"
Rectangle2D addIniModule: [ :point | IniModule nextIni: { #x -> (point x).
                                                          #y -> (point y) } ].

"Rectangle2D>>(width)()
initialize width"
Rectangle2D addRequiredIniModule: [ :width :instVars | instVars at: #width put: width.
                                                       IniModule nextIni: {  } ].

"Rectangle2D>>(height)()
initialize height"
Rectangle2D addRequiredIniModule: [ :height :instVars | instVars at: #height put: height.
                                                        IniModule nextIni: {  } ]
```

Figure 4.1: Ini-modules for the class `Rectangle2D`

The input parameters of each ini-module directly correspond to the arguments of the block (except for the special argument `instVars`, see below). Ini-modules are modular, this means that they cooperate. In this example, `Rectangle2D»(angle rad)(x y)` and `Rectangle2D»(point)(x y)` indirectly delegate to `Rectangle2D»(x y)()` by using the parameter names.

The messages `addIniModule:` and `addRequiredIniModule:` take as argument a block that:

- may contain a special argument `instVars` to initialize the fields, by sending a message `at:put:` to it; notice that this argument is *not* an input parameter of the ini-module;
- must contain a sent of the message `nextIni:` to the class `IniModule`; the argument of this message is a dynamic array, enclosed between `{` and `}`, and containing associations mapping each output parameter to a value (the array is empty when there are no output parameters).

For example, the `Rectangle2D»(x y)()` ini-module has no output parameters, therefore the dynamic array parameter of `nextIni:` is empty. Its input parameters are `x` and `y`, which are used to set the homonymous fields. The `Rectangle2D»(angle rad)(x y)` ini-module takes two input parameters, `angle` and `rad`, and produces two output parameters, `x` and `y`. Therefore, the dynamic array parameter of `nextIni:` contains two associations created using the message `->`, both calculating the value for the output parameters `x` and `y`, and associating these calculated values to them.

We recall that ini-modules do not allow the class designer to read field values, while field assignments are possible, through the special argument `instVars`. This is consistent with the

```
"ColoredRectangle2D>>(red green blue)()"
ColoredRectangle2D addRequiredIniModule: [ :red :green :blue :instVars |
                                            instVars at: #r put: red;
                                                     at: #g put: green;
                                                     at: #b put: blue.
                                            IniModule nextIni: {  } ].

"ColoredRectangle2D>>(c m yc k)(red green blue)"
ColoredRectangle2D addIniModule: [ :c :m :yc :k |
                                   IniModule nextIni: { #red -> (255 * (1 - c) * (1 - k)).
                                                        #green -> (255 * (1 - m) * (1 - k)).
                                                        #blue -> (255 * (1 - yc) * (1 - k)) } ]
```

Figure 4.2: Ini-modules for the subclass `ColoredRectangle2D` of `Rectangle2D`

model we presented in Chapter 2, where we stated as requirement to prohibit field access in the body of an ini-module as it may introduce an implicit dependency between ini-modules. Instead, we want all the dependencies between ini-modules to be explicit in their signatures.

In the object-creation expression of the Pharo implementation, the new-supplied parameters are provided as an array of associations passed as argument to the message `mipNew:`. This message is sent to the class from which the developer wants to create an instance (in this case, `Rectangle2D`):

```
|rectangle|
rectangle := Rectangle2D mipNew: { #point -> (5@20).
                                   #width -> 50.
                                   #height -> 10 }
```

The `mipNew:` method implements the function CreateInstance as shown in Algorithm 1.

4.1.1 Extending the initialization protocol in subclasses

We show now how to implement in Pharo the subclass `ColoredRectangle2D` of `Rectangle2D`, with its three new fields `r`, `g` and `b` for color:

```
Rectangle2D subclass: #ColoredRectangle2D
  instanceVariableNames: 'r g b'
  classVariableNames: ''
  category: 'IniModules-Tests-Classes'
```

To manage two different color palettes (RGB and CMYK), we only need to add two new ini-modules, as shown in Figure 4.2. The `ColoredRectangle2D»(red green blue)()` ini-module takes three input parameters and assigns them to the newly introduced fields, by using the `instVars` special argument of the block. The `ColoredRectangle2D»(c m yc k)(red green blue)` ini-module calculates the values to be assigned to its output parameters by starting from its input parameters.

For example, to create a purple rectangle:

```
|purpleRectangle|
purpleRectangle := ColoredRectangle2D mipNew: { #point -> (5@20).
                                                #width -> 50.
                                                #height -> 10.
```

```
#c -> 0.5.
#m -> 0.8.
#yc -> 0.3.
#k -> 0.1 }
```

Note that the ini-modules defined through the hierarchy of the instantiated class `ColoredRectangle2D` are not ordered by any explicit constraint. Therefore, only the default constraints contribute to determine the total ordering in which ini-modules are considered for activation. Assuming a total ordering of the ini-modules as shown in Figure 4.3, the evaluation of this message generates the following chain of activations:

1. ini-modules are looked for starting from the instantiated class and going up in the class hierarchy; the first class to be considered is then `ColoredRectangle2D`, where the `ColoredRectangle2D»(c m yc k)(red green blue)` ini-module is activated by the new-supplied parameters `c`, `m`, `yc` and `k`, which are therefore terminated; `red`, `green` and `blue` are parameters produced by this ini-module, which are added to the parameter map;

2. then, the `ColoredRectangle2D»(red green blue)()` ini-module is activated, terminating three parameters from the parameter map, setting the `r g b` fields, and outputting nothing;

3. going up in the hierarchy, the class `Rectangle2D` is encountered; its first ini-module to be considered is `Rectangle2D»(angle rad)(x y)`, which is discarded because `angle` and `rad` are not part of the parameter map, which contains values for `point`, `width` and `height`;

4. the `Rectangle2D»(point)(x y)` ini-module is activated, terminating `point` and module-supplying `x` and `y`;

5. the ini-modules `Rectangle2D»(x y)()`, `Rectangle2D»(width)()` and `Rectangle2D»(height)()` are activated one after the other, each terminating one parameter and outputting nothing;

6. finally, the ini-module `Object»()()` is activated, doing nothing;

7. with no more ini-modules and an empty parameter map, the object creation returns successfully the newly created object.

4.1.2 How to specify explicit constraints

In this section we show how the class designer can force an ordering between ini-modules by defining explicit constraints. Ini-modules in Pharo are objects, indeed the two messages `addIniModule:` and `addRequiredIniModule:` return instances of the class `IniModule`. This means that it is possible to assign an ini-module to a variable, giving it a name. Such variables

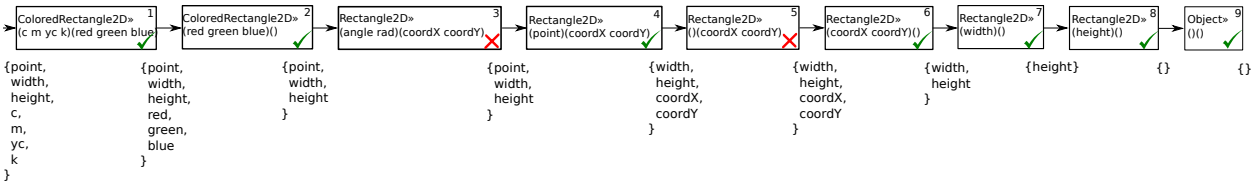

Figure 4.3: Execution of an object-creation expression (the parameter map shows parameter names only, not values).

can then be used to define explicit ordering constraints. This makes the definition of an explicit constraint partially less verbose than in the model, even though, as we shall see, it is sometimes necessary to retrieve a previously defined ini-module by means of the message `iniModuleWith:`.

To define an explicit constraint, the class designer can send a message `applyBefore:` or `applyAfter:` to an ini-module, with the other ini-module to put in an order relationship with the receiver passed as argument.

We show now how to initialize in Pharo the instances of the example class `Person`, to obtain the desired behavior:

```
Object subclass: #Person
  instanceVariableNames: 'name surname nickname code'
  classVariableNames: ''
  package: 'IniModules-Tests-Classes'
```

We recall from Section 3.2 that:

- if no nickname is submitted in an object-creation expression, then we want a default one to be automatically generated from the name and the surname;
- every person is associated to a code, and we want this code to be calculated starting from nickname (if provided) or name/surname (if not).

In Section 3.2 we have already seen the ini-modules for the class `Person` (Figure 3.3). Figure 4.4 shows how to define them in Pharo:

- `(nick)(nick, code)` (line 2) creates a code starting from a nickname; note that we need to put the created ini-module into a local variable `codeFromNick` to later define the explicit constraint;
- `(name, surname)(name, surname, code)` (line 8) creates a code starting from name and surname; we put also this ini-module into a local variable `codeFromName`;
- the explicit constraint is declared by sending the message `applyBefore:` to the object `codeFromNick`, with argument `codeFromName` (line 15);

```
1  |codeFromNick codeFromName|
2  codeFromNick := Person addIniModule:
3                          [ :nick | IniModule nextIni: {
4                                      #nick -> nick.
5                                      #code -> ('a code from ', nick)
6                                    }
7                          ].
8  codeFromName := Person addIniModule:
9                          [ :name :surname | IniModule nextIni: {
10                                            #name -> name.
11                                            #surname -> surname.
12                                            #code -> ('a code from ', name, ' ', surname)
13                                          }
14                         ].
15 codeFromNick applyBefore: codeFromName.
16 Person addIniModule: [ :name :surname | IniModule nextIni: {
17                                           #name -> name.
18                                           #surname -> surname.
19                                           #nick -> ('a nickname from ', name, ' ', surname)
20                                         }
21                      ].
22 Person addRequiredIniModule: [ :name :surname :instVars | instVars at: #name
23                                                                 put: name;
24                                                                 at: #surname
25                                                                 put: surname.
26                                                          IniModule nextIni: {} ].
27 Person addRequiredIniModule: [ :nick :instVars | instVars at: #nick put: nick.
28                                                 IniModule nextIni: {} ].
29 Person addRequiredIniModule: [ :code :instVars | instVars at: #code put: code.
30                                                 IniModule nextIni: {} ]
```

Figure 4.4: How to define the ini-modules for the class `Person` in Pharo.

- `(name, surname)(name, surname, nick)` (line 16) creates a default nickname starting from name and surname; note that we do not need anymore to save the created ini-modules into local variables, as we do not have other explicit constraints to define;
- the last three ini-modules set the respective fields.

4.1.3 Graphical user interface for ini-modules

We have prototyped a graphical user interface for ini-modules in Pharo. Our prototypical GUI integrates ini-module visualization and editing into the System Browser, which is Pharo's browser for browsing packages, classes, protocols, methods and method definitions. We present it by using the previously introduced example of `Person`.

Once a class has been created, a protocol '-- ini-modules --' appears by default on the third panel of the System Browser (Figure 4.5). This panel usually shows all the protocols for the selected class, which in Pharo are convenient groupings of related methods. We extended it with a protocol '-- ini-modules --', which is thus simply a label for a grouping containing all the ini-modules. By clicking on it:

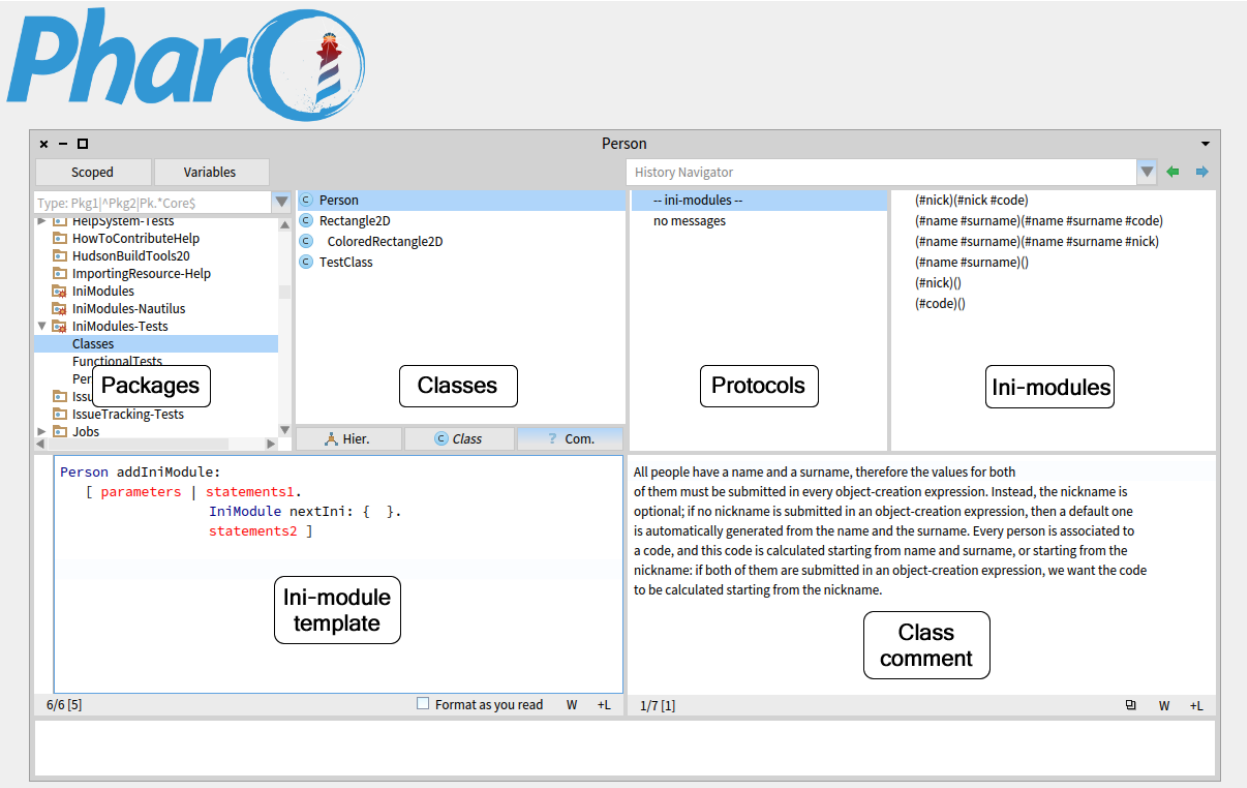

Figure 4.5: Pharo's System Browser as extended for ini-modules.

- the underlying panel shows a template for ini-module definition, guiding the class designer in defining a new ini-module (the message `addIniModule:` must be changed into `addRequiredIniModule:` if the class designer wants to create a new required ini-module);

- the fourth panel shows, via their signatures, the list of the defined ini-modules for the selected class; in this panel, the shown ini-modules are ordered by the total ordering given by the inferred default constraints and the imposed explicit constraints; this allows the class designer to have a direct check over the ordering in which the ini-modules are considered for activation.

By clicking on one of the ini-modules in the list, the panel below shows the ini-module definition code (Figure 4.6), allowing the user to edit it when needed. If explicit constraints involving the shown ini-module have been declared, they are shown as well. When defining an explicit constraint by using the underlying panel of the System Browser, the message `iniModuleWith:` can be sent to a class for retrieving a previously defined ini-module. This message takes as argument the signature of the ini-module to retrieve, as a literal array containing two nested arrays of symbols. For example, in Figure 4.6 the message-send:

```
Person iniModuleWith: #((name surname)(name surname code))
```

retrieves the ini-module `Person»(name surname)(name surname code)`. Note that the above notation for referring to the ini-module signature is equivalent, in Pharo, to `#(#(#name #surname) #(#name #surname #code))`, but more concise. Therefore, the `iniModuleWith:` message takes as argument a literal array with inside two nested arrays containing symbols.

Finally, by right-clicking on an ini-module in the fourth panel of the System Browser, a drop-down menu appears allowing the user to remove it.

4.2 Ini-modules as methods

At first, we implemented ini-modules in Pharo as methods and we marked them with an appropriate annotation:

```
TheClass>>iniModuleWithInput1: input1 ... inputN: inputN
  <initializerWithInput: #(input1  ... inputN) withOutput: #(output1 ... outputN)>
  ...
  ^ { aValueForOutput1. ... aValueForOutputN }
```

In Pharo an annotation is called a *pragma* and can be applied to a method declaration only. An annotation is declared before any statement and represents a static expression with literal arguments, which can be examined at compile time and runtime. The `initializerWithInput:-withOutput:` pragma marks a method as being an ini-module. The pragma has 2 arguments, its input and output parameters as arrays of names.

The body of an ini-module is supposed to always end by returning a dynamic array (enclosed between `{` and `}`), in such a way that a tool can statically check that the returned

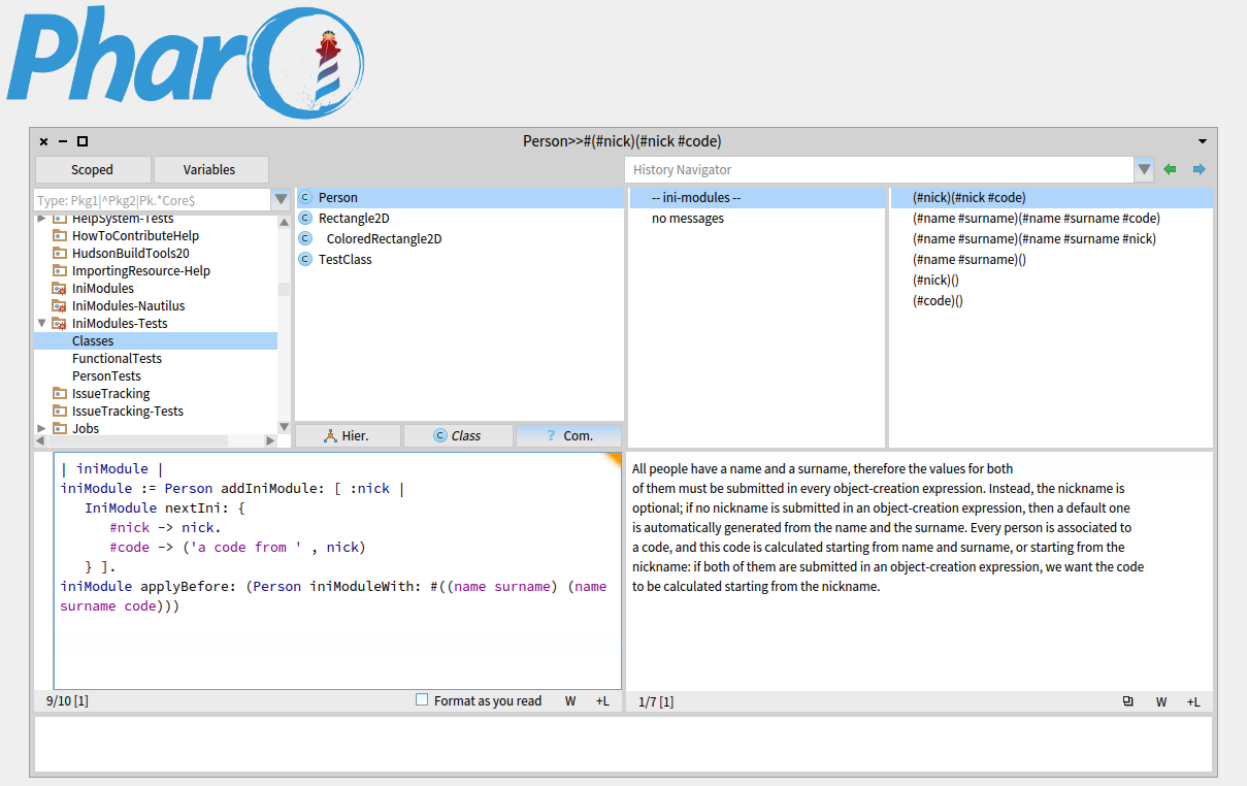

Figure 4.6: Visualization and editing of an ini-module.

array's size matches the number of output parameters. This dynamic array contains a value for each of the output parameters.

Figure 4.7 presents how the ini-modules for the `Rectangle2D` and `ColoredRectangle2D` classes can be defined as methods marked with annotations.

4.2.1 Drawbacks of the discussed approach

We started by representing ini-modules as methods because a method gives the ini-modules direct write access to class fields, thus satisfying Requirement 2 (Direct access to object fields)[1].

Nevertheless, methods are too powerful for ini-modules, therefore they permit forbidden behavior: (1) an ini-module can directly call ini-modules using message sends, thus bypassing the general activation algorithm defined in Section 2.3; (2) ini-modules can be called outside the object initialization phase, this means that Requirement 4 (Constructors activation context) is not satisfied; (3) ini-modules can also read instance fields, in contravention of Check 7.

[1]See Section 1.1.2

```
Rectangle2D>>x: anInt y: anotherInt
  <initializerWithInput: #(x y) withOutput: #()>
  "initialize lower-left corner from Cartesian coordinates"
  x := anInt.
  y := anotherInt.
  ^ {  }

Rectangle2D>>angle: angle rad: rad
  <initializerWithInput: #(angle rad) withOutput: #(x y)>
  "initialize lower-left corner from polar coordinates"
  ^ { angle cos * rad . angle sin * rad }

Rectangle2D>>point: aPoint
  <initializerWithInput: #(point) withOutput: #(x y)>
  "initialize lower-left corner from a Point object"
  ^ { aPoint x . aPoint y }

Rectangle2D>>width: anInt
  <initializerWithInput: #(width) withOutput: #()>
  "initialize width"
  width := anInt.
  ^ {  }

Rectangle2D>>height: anInt
  <initializerWithInput: #(height) withOutput: #()>
  "initialize height"
  height := anInt.
  ^ {  }

ColoredRectangle2D>>r: red g: green b: blue
  <initializerWithInput: #(red green blue) withOutput: #()>
  r := red.
  g := green.
  b := blue.
  ^ {  }

ColoredRectangle2D>>c: anInt1 m: anInt2 yc: anInt3 k: anInt4
  <initializerWithInput: #(c m yc k) withOutput: #(red green blue)>
  |red green blue|
  red := (255 * (1 - anInt1) * (1 - anInt4)).
  green := (255 * (1 - anInt2) * (1 - anInt4)).
  blue := (255 * (1 - anInt3) * (1 - anInt4)).
  ^ { red. green. blue }
```

Figure 4.7: Ini-modules as methods for the classes `Rectangle2D` and `ColoredRectangle2D`

Moreover, the ini-module model described in Chapter 2 specifies `I2`, a sequence of instructions, possibly empty, executed after the invocation of other ini-modules. If ini-modules are implemented as methods, the return statement (caret `^`) is used to tell the invocation algorithm the currently executing ini-module is finished. As a consequence, `I2` instructions are not allowed in this implementation.

Chapter 5

Evaluation

In this chapter, we discuss how our approach to object initialization meets the requirements that we outlined in Chapter 1.

5.1 Solving the initialization problems of Pharo

With respect to the specific problems of Pharo initialization approach, we observe that:

Requirement 1 (Parameter passing) The MIP mechanism allows the developer to define ini-modules both to deal with default parameters and with parameters supplied at object creation time.

Requirement 2 (Direct access to object fields) The MIP mechanism does not require the class designer to implement mutating methods because ini-modules can directly modify the class fields.

Requirement 3 (No pollution of the class' API) The MIP mechanism does not pollute the class' API, provided that ini-modules are implemented as objects (Section 4.1) and not as methods (Section 4.2).

Requirement 4 (Constructors activation context) Ini-modules are not callable from outside the initialization process.

Requirement 5 (Complete control over the initialization process) The MIP mechanism makes the `initialize` method useless to initialize fields and thus no default initialization happens if the class user specifies initialization values.

```
"ColoredPoint>>(red green blue)()"
ColoredPoint addRequiredIniModule: [ :red :green :blue :instVars |
                                     instVars at: #r put: red;
                                              at: #g put: green;
                                              at: #b put: blue.
                                     IniModule nextIni: { } ].

"ColoredPoint>>(c m yc k)(red green blue)"
ColoredPoint addIniModule: [ :c :m :yc :k |
                              IniModule nextIni: { #red -> (255 * (1 - c) * (1 - k)).
                                                   #green -> (255 * (1 - m) * (1 - k)).
                                                   #blue -> (255 * (1 - yc) * (1 - k)) } ]
```

Figure 5.1: `ColoredPoint` class refactored with ini-modules.

5.2 Solving the number of constructors explosion

5.2.1 Multiple initialization options

In Section 1.2.1 we stated Requirement 6 regarding the multiple initialization options explosion. Figure 1.1 shows how the six `ColoredPoint` instance-creation methods would be implemented in a standard Pharo.

With MIP, each initialization protocol of a field set is managed by a dedicated ini-module. As a result, the developer avoids duplicating code. For example, the `ColoredPoint` class only needs two ini-modules (Figure 5.1) instead of six instance-creation methods (Figure 1.1). Initializing the `x` and `y` fields is now responsibility of only `Point2D`. No more duplication is required.

5.2.2 Optional initialization

In Section 1.2.2 we stated Requirement 7 regarding the optional initialization explosion. We saw that the `TextArea` class implemented in standard Pharo would require the class designer to implement 7+1 instance creation methods (Figure 1.2), at the aim of providing all the initialization options.

With MIP, for each optionally initialized field, we must implement two ini-modules:

- the first ini-module has no input parameters and outputs the default value for that field;
- the second ini-module takes as input a value for the optionally initialized field and has no output parameters.

Figure 5.2 shows the `TextArea` example factored out with ini-modules. Also in this case, the number of ini-modules is linear in the number of optional fields: only six ini-modules (compared to seven instance-creation methods normally required in Figure 1.2) are required. Moreover, each ini-module takes care of a single initialization protocol. No more duplication is required.

5.2.3 Problems with subclassing

In Chapter 1.2.3 we stated Requirement 8 regarding duplication in subclasses. In standard Pharo, for an `OnOffTextArea` subclass of the `TextArea` class, which adds a `disabled` boolean field to indicate whether the text area is enabled or not, the five instance-creation methods of the superclass must be duplicated in `OnOffTextArea`. Additionally, if specifying the initial value of the `disabled` field is optional, the `OnOffTextArea` must implement 10 instance-creation methods.

With MIP, the `OnOffTextArea` developer just needs to implement a single ini-module and two if initializing `disabled` is optional (as can be seen in Figure 5.3): only two ini-modules (compared to ten instance-creation methods) are required now. No more duplication is required. Note that specifying ordering of these two modules is not necessary as the partial ordering algorithm discussed in Section 3.2 makes sure the `OnOffTextArea»()(disabled)` ini-module is considered first.

Stop delegating initialization to subclasses and clients

As we have just seen, ini-modules make it possible to extend the superclass initialization protocol without duplicating the constructors of the superclass in the subclass. Sometimes, the developer of a subclass may want to impose a particular value on a superclass initialization protocol and prevent client code from specifying a different value [15]. Ini-modules also make this possible. Let us consider a class `Bird` with a boolean field `flies`, and a subclass `Penguin` which wants to set the field `flies` to `false`. The class designer can define the following set of ini-modules to obtain the desired behavior:

```
"Bird>>(flies)()"
Bird addRequiredIniModule: [ :flies :instVars | instVars at: #flies put: flies.
                                                IniModule nextIni: { } ].

"Penguin>>(flies)()"
Penguin addIniModule: [ :flies :instVars | "do nothing, only eat the parameter"
                                           IniModule nextIni: { } ].

"Penguin>>()(flies)"
Penguin addRequiredIniModule: [ | IniModule nextIni: { #flies -> false } ]
```

This combination of ini-modules makes sure that (1) if `flies` is provided in an object-creation expression instantiating the class `Penguin`, it will be consumed and ignored by the ini-module `Penguin»(flies)()`; (2) in all cases, `flies` will be set to `false` by `Penguin»()(flies)` which is activated after `Penguin»(flies)()` (note that the default Rule 2 applies here).

Note that we declared the ini-module `Penguin»()(flies)` as required because this makes sure that an object-creation expression instantiating the class `Penguin` does not supply the parameter `flies`. In fact, in such a case the ini-module `Penguin»()(flies)` would not be activated, causing a runtime error (we recall from Section 2.3 that a runtime error is raised when a required ini-module is not activated).

```
Object subclass: #TextArea
  instanceVariableNames: 'rows columns text scrollbars'

"TextArea>>()(rows columns)"
TextArea addIniModule: [ | IniModule nextIni: { #rows -> 0. #columns -> 0 } ].

"TextArea>>()(text)"
TextArea addIniModule: [ | IniModule nextIni: { #text -> '' } ].

"TextArea>>()(scrollbars)"
TextArea addIniModule: [ | IniModule nextIni: { #scrollbars -> Scrollbars both } ].

"TextArea>>(rows columns)()"
TextArea addRequiredIniModule: [ :rows :columns :instVars |
                                    instVars at: #rows    put: rows;
                                             at: #columns put: columns.
                                    IniModule nextIni: { } ].

"TextArea>>(text)()"
TextArea addRequiredIniModule: [ :text :instVars |
                                    instVars at: #text put: text.
                                    IniModule nextIni: { } ].

"TextArea>>(scrollbars)()"
TextArea addRequiredIniModule: [ :scrollbars :instVars |
                                    instVars at: #scrollbars put: scrollbars.
                                    IniModule nextIni: { } ]
```

Figure 5.2: The `TextArea` class refactored with ini-modules.

```
TextArea subclass: #OnOffTextArea
  instanceVariableNames: 'disabled'

"OnOffTextArea>>()(disabled)"
OnOffTextArea addIniModule: [ | IniModule nextIni: { #disabled -> false } ].

"OnOffTextArea>>(disabled)()"
OnOffTextArea addRequiredIniModule: [ :disabled :instVars |
                                        instVars at: #disabled put: disabled.
                                        IniModule nextIni: { } ]
```

Figure 5.3: The `OnOffTextArea` class refactored with ini-modules.

Chapter 6

Related work

The work presented in this thesis is related to other work concerning both dynamically and statically-typed languages.

6.1 Common Lisp / CLOS

In the Common Lisp Object System, a field declaration also includes information about the default value and how to initialize the field with a user-provided value. For example, for a class definition such as this one:

```
(defclass textarea ()
  ((columns    :initarg  :columns    :initform 50))
  ((rows       :initarg  :rows       :initform 4))
  ((text       :initarg  :text       :initform ""))
  ((scrollbars :initarg  :scrollbars :initform 'both))
  ((a-priv-field                     :initform 42)
```

the following is a valid object-creation expression:

```
(make-instance 'textarea :columns 80 :text "Initial text")
```

This expression instantiates the `textarea` class with 80 columns, an initial text, and default values for the number of rows (4) and scrollbar type (the `both` type). As a result of this form of field declaration, CLOS makes it easy to define which field can be given a value at initialization time by the user (through `:initarg`) and which field has default values (through `:initform`). With this in place, CLOS meets all but Requirement 6 (Multiple initialization options). A workaround is using the CLOS meta-object protocol: for example, a developer could implement the Algorithm 1 in a piece of advice for the `make-instance` method.

6.2 Constructors and methods with named and optional parameters

Constructors with *default* parameters (present, for instance, in Delphi and C++) make it possible to declare fewer constructors. For example, in C++, a `point` class could be defined

as follows:

```
class point
{
private:
    int columns_;
    int rows_;
public:
    point(int columns = 0, int rows = 0) :
        columns_(columns),
        rows_(rows)
    {
    }
};
```

This code defines a constructor with two optional arguments for `x` and `y`. This class can be instantiated with either of these two lines:

```
point p1;
point p2(10,20);
```

If Pharo had such a feature, we could rewrite the `TextArea` instance-creation methods using just one method as follows:

```
TextArea class>>new(columns: 0) (rows: 0)
                  (text: '') (scrollbars: #both)

  ^ self basicNew
      initialize;
      columns: cols;
      rows: rows;
      text: text;
      scrollbars: scrollbars;
      yourself
```

This imaginary code snippet would be equivalent to the manual definition of 16 constructors. Even though this optional argument feature would meet Requirement 7 (Optional initialization), it would not meet all the other requirements.

Another problem with this approach is that each specification of a default value for a field appears in all constructors that allow an optional specification of a value for this field.

6.3 Other approaches to initialization

Avoiding the initialization protocol. The idea is to have only one parameterless constructor (or none) and performing the actual initialization by invoking mutating (*i.e.,* setter) methods.

The drawback with this approach is that there is no direct way to check if an object is initialized, as it is the class user's responsibility to call the right methods in the correct order. This can be mitigated by the use of formal specification languages to describe the API protocol.

Container classes. A *container class* is a class whose objects are used for encoding the parameters needed to initialize field sets in another class. A container class must have as many constructors as the possible options of initialization of the field set.

The `ColoredPoint` example (see Figure 1.1) written using this pattern would look as follows:

```
class Position // container class
{ private float x, y;

  public Position(float x, float y){...};
  public Position(Complex comp){...};
  public Position(float angle, float rad, boolean polardef){...};
  ...
}

class Color // container class
{ private float r, g, b;

  public Color(float r, float g, float b) {...};
  public Color(float c, float m, float y, float k) {...};
  ...
}

class ColoredPoint
{ private Color c;
  private Position p;

  public ColoredPoint(Color c, Position p){...};
  ...
}
```

This approach meets Requirement 6 (Multiple initialization options), avoiding related code duplication. Nevertheless, its application must be foreseen in the earlier stages of a class design.

Constructor propagation in Java Layers. The Java Layers language [7] has a feature called *constructor propagation*, which we explain by an example:[1]

```
class Position
{ propagate Position(float x, float y)  {I_1}
  propagate Position(float angle, float rad)  {I_2}
}
class Color<T> extends T
{ propagate Color(float r, float g, float b)  {I_3}
  propagate Color(float c, float m, float y, float k)  {I_4}
}
```

As the result of the above definition, the `Color<Position>` class includes the same constructors as if it was defined like this:

```
class Color {
  Color (float x, float y,
           float r, float g, float b) {I_1; I_3}
  Color (float x, float y,
           float c, float m, float y, float k) {I_1; I_4}
  Color (float angle, float rad,
           float r, float g, float b) {I_2; I_3}
  Color (float angle, float rad,
           float c, float m, float y, float k) {I_2; I_4}
  ...
}
```

[1] Inspired by an example taken from http://www.cs.utexas.edu/~richcar/cardoneDefense.ppt

This approach meets Requirement 6 (Multiple initialization options) in most cases, but not always. Indeed, with Java Layers it is not possible to implement *a posteriori* a subclass of a `C` class that has the sole purpose of adding a new option of initialization for a field declared in `C`. For example, adding an `HSB` extra palette to the `ColoredPoint` class (Figure 1.1) without modifying its code requires duplicating all constructors in a subclass. Doing the same in Pharo with ini-modules only requires adding one subclass containing one ini-module.[2]

Template constructors. Martin *et al.* [15] proposed *template constructors* enabling a one-to-many binding of super-calls:

```
class ColoredPoint extends Point {
  private float r, g, b;
  public ? ColoredPoint(p*, float r, float g, float b) {
    super(p*);
    this.r = r;
    this.g = g;
    this.b = b;
  }
}
```

This constructor uses a *template parameter* `p*` in its list of formal parameters and a *template argument* `p*` in the super-call in its body. The template constructor will accept each list of arguments `p*` followed by three floats, such that there is a superclass constructor accepting the list `p*`. Since `super(p*)` matches all superclass constructors, one `ColoredPoint` concrete constructor (a standard Java constructor) will be generated for each constructor of `Point`, at the moment of loading the class. Template constructors also support *named expression arguments* in super-calls:

```
class Penguin extends Bird {
  public ? Penguin(p*) { super(p*, flies : false); }
}
```

A named expression argument only matches a formal parameter with exactly the same name, ignoring the position of the named expression and the matched parameter. This feature allows a subclass to set a certain parameter of a superclass constructor, similarly to what we showed for ini-modules in Section 5.2.3. Furthermore, template constructors address the lack of support for mixin constructors, which was one of the motivations behind the introduction of ini-modules in Magda [13, 4].

Published after Magda, template constructors meet Requirements 6 (Multiple initialization options) and 8 (Inheritance of the initialization protocol (no signature duplication)). The proposed solution to the problem of optional initialization seems to contradict their constructor generation algorithm, therefore we are unable to tell whether template constructors meet Requirement 7 (Optional initialization). Furthermore, applying the main idea behind template constructors (that is, enabling superclass constructors in subclasses) to Pharo, where constructors are already inherited, would not solve all the initialization problems of Pharo, in particular it would not meet Requirement 2 (Direct access to object fields).

[2]With class extensions, an ini-module (or any method) can even be added to an existing class belonging to another package, thus no subclass is required.

In general, we believe that our approach is more ambitious since ini-modules represent a completely new design of initialization protocols, whereas template constructors are more practical since they can easily be used alongside the Java TIP.

6.4 Other works related to initialization

Field initialization analysis. Object initialization is crucial for proving class invariants. Usually, in modern programming languages, fields of objects hold a default value, contrary to local variables which are required to be explicitly initialized before their use. Proving that a property p holds for a field f can not be limited to prove that all assignments to f write a value with some property p. Indeed, it is also needed to prove that f is always initialized before being read.

An object is called *raw* [10] when its fields are not all initialized yet. This means that such an object may violate its object invariants, for example that a given field is non-`null`. An object loses its rawness as soon as all its fields have been initialized.

Spoto *et al.* [19] propose a static *initialization analysis* that infers a safe over-approximation of the set of local variables, fields, parameters, return values, and array elements that, at runtime, might hold raw objects. They implemented their static analysis in a tool called Julia [18], that computes sound and precise rawness and nullness annotations. Julia works over Java bytecode, but it can be applied to Java source code as well, by compiling it into bytecode. Nullness analysis determines, at compile-time, those program points where the `null` value might be dereferenced, leading to a run-time exception. Initialization analysis is important even for a field that is intended to be able to hold `null`, because there are usually object invariants that relate field values to one another, *e.g.,* that exactly one out of two fields should be `null`, or that one field's value is meaningful only if another field is non-`null`. Moreover, initialization analysis can also prevent a programmer for violating object invariants by forgetting to initialize a field, and can ensure that a programmer explicitly initializes fields to `null` when appropriate.

In Section 2.5.3 we outlined how the ini-modules approach relates to class invariants. As we pointed out in Section 2.5.2, we decided not to force in Pharo the initialization of all the fields of a newly created object (see Check 6). This design choice is due to the widespread use of lazy initialization among Pharo developers. An initialization analysis similar to that conducted in [19] may be carried out on Pharo code bases to help developers proving class invariants.

X10. Zibin *et al.* [21] propose X10, an object-oriented programming language with a sophisticated type system (constraints, class invariants, non-erased generics, closures) and concurrency constructs. They identify a set of properties that a design for object initialization should have. Among these we have the followings (we omit those related to concurrency and type-safety):

- **Cannot read uninitialized fields** One should not be able to read uninitialized fields.

- **Single value for final fields** Final fields can be assigned exactly once, and should be read only after assigned.
- **Simple** The order of initialization should be clear from the syntax, and should not surprise the user. Dynamic dispatch during construction disrupts the order of initialization by executing a subclass' method before the superclass finished its initialization. This kind of initialization order is error-prone and often surprises the user.
- **Flexible** The user should be able to express the common idioms found in other languages with minor variations.

X10 is based on an hardhat design [11], which is a design that prohibits dynamic dispatch or leaking `this` (*e.g.,* storing `this` in the heap) during construction. Dynamic dispatch may transfer control to the subclass before the superclass completed its initialization.

A comparison with X10 may help us in our investigation concerning field access in ini-modules bodies (see Check 7 in Section 2.5.2). Moreover, we think that the ini-modules approach is sufficiently simple and flexible. However, as we outlined in Section 3.2.2, the total ordering that is generated by our algorithm is currently not predictable and this is a point that may be improved to make more straightforward for the user the order in which ini-modules are activated.

Chapter 7

Future work

7.1 Extending our analysis to other dynamic languages

This work originates from work on the statically-typed language Magda. It adapts Magda's ini-modules initialization approach to the dynamically-typed language Pharo.

A promising future research direction is to perform a full study of object initialization approaches in many dynamic languages, along with an analysis of their limitations. To do so, a significant pool of dynamic languages shall be firstly identified. This pool should include at least CLOS, Python, Ruby, Julia, Grace and JavaScript. Also alternative or more general approaches to initialization, such as template constructors or constructors with named and optional parameters (see Chapter 6) should be considered and included in the pool of the various approaches to initialization, to be compared to ini-modules.

A requirement matrix can then be constructed, with a column for each initialization approach (language-specific or not), and a row for each of the requirements identified in Chapter 1. Each cell of the matrix corresponds to the crossing of an initialization approach and a requirement, and it is checked if and only if that initialization approach meets that requirement.

A work that is directly connected to the study of a pool of dynamic languages initialization approaches is to clearly define a terminology to distinguish between different approaches to initialization. Indeed, some languages have (1) dedicated syntactic constructs, some other have (2) special versions of general constructs (*e.g.,* methods) that are, in some way, reserved for initialization (the relative code is automatically executed during object creation and it is intended to be executed only during object creation); or again, some languages rely on (3) workarounds, *i.e.,* on features of the language that are in no way, neither syntactically nor in any other way, reserved for initialization. Actually, a combination of all these approaches can be found in many dynamic languages.

For example, we have seen in Section 1.1 that Pharo relies on the `initialize` method, which is an example of (2), and on instance-creation methods and lazy initialization, which are examples of (3).

Similarly to Pharo, Python has a special method named `__init__` which is used to

initialize the object fields, and that is automatically invoked during instantiation from a class. Methods like `__init__` are called *magic methods* in Python, because they are special methods that you can define to add "magic" to the classes; they are always surrounded by double underscores. Here is a Python class `OrderedPair`:

```
>>> class OrderedPair:
...   def __init__(self, first, second):
...     self.f = first
...     self.s = second
...
>>> p = OrderedPair('first', 'second')
>>> p.f, p.s
('first', 'second')
```

Python's `__init__` method is another example of (2).

Instead, Julia has a dedicated syntactic construct for initialization. A constructor can be declared inside the block of a type declaration:

```
>>> type OrderedPair
>>>   x::Real
>>>   y::Real
>>>
>>>   OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
>>> end
>>>
>>> myPair = OrderedPair(5,20)
OrderedPair(5,20)
```

The constructor `OrderedPair(x,y)` is an example of (1).

7.2 Maximising ini-modules reuse

Different classes may duplicate one or more sets of ini-modules, when they share common properties. For example, in Chapter 2 we considered both classes `Point2D` and `Rectangle2D`. Looking at the ini-modules they define for initializing the fields `x` and `y`, one can notice that the three ini-modules `(angle, rad)(coordX, coordY)`, `(point)(coordX, coordY)` and `()(coordX, coordY)` are exactly duplicated in the two classes. As a future work, we plan to consider possible solutions to make it possible for a class to import the ini-modules of another one, with the aim of maximising code reuse. In particular, we will investigate the impact of ini-modules on both stateless [9] and stateful traits [2].

7.3 Performance

Another future work is to investigate ini-modules performance in Pharo. For this purpose, we plan to identify meaningful libraries in Pharo to be refactored with ini-modules. Identifying initialization code in Pharo projects is not easy at all because, as we have seen in Section 1.1, Pharo developers usually rely on common patterns to initialize object fields (see also [1]). As a consequence, the initialization code of a Pharo class can be scattered in different places: for example, instance-creation methods are implemented on the class side and lazy initialization

is implemented on the instance side, into accessors. The Moose platform[1] for software and data analysis is based on Pharo and may turn out to be useful in this context. Indeed, we plan to query substantial code bases of existing Pharo projects with the aim of identifying some libraries that would be interesting to refactor with ini-modules. For example, the Pharo catalog[2], a list of registered Pharo projects, can be used for this purpose. The Pharo catalog is not exhaustive but it shows some of the most important community projects around. Then, executing a same set of object-creation expressions, in the original libraries and in those refactored with ini-modules, will give us an indication on how much performing ini-modules are compared to the traditional Pharo initialization. Moreover, we will compare simplicity and concision of source code as well as real-world developers acceptance and appreciation.

7.4 Going beyond the limitations of the prototype

We have seen in Chapter 4 that our Pharo implementation relies on the theory we exposed in Section 3.2 regarding the ordering approach, allowing the class designer to impose a partial ordering of the defined ini-modules. We pointed out in Section 3.2.2 that a crucial point of the ordering algorithm is how to force the determinism of the obtained total ordering. For the first prototype, we skipped this point, that means that the computed total ordering is not guaranteed to be deterministic. As we said, the determinism can be easily met by using some deterministic criterion for selecting the node to be considered at each step, *e.g.,* by using a hash function that is deterministic and perfect. In Pharo, such a function may be represented by the message send `aBlock sourceNode hash`, where `aBlock` is the block used for defining an ini-module. The `hash` message returns a number based on the tree representation of the block code. We have to check if this number is always the same for a given block in different images on different computers.

[1] http://www.moosetechnology.org/
[2] http://catalog.pharo.org/

Bibliography

[1] K. Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC'06)*, volume 4406 of *LNCS*, pages 66–90. Springer, Aug. 2007.

[3] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[4] V. Bono, J. Kuśmierek, and M. Mulatero. Magda: A new language for modularity. In *ECOOP*, pages 560–588, 2012.

[5] V. Bono and J. D. M. Kusmierek. FJMIP: A calculus for a modular object initialization. In *Fundamentals of Computation Theory, 16th International Symposium, FCT 2007, Budapest, Hungary, August 27-30, 2007, Proceedings*, pages 100–112, 2007.

[6] V. Bono and J. D. M. Kuśmierek. Modularizing constructors. *Journal of Object Technology*, 6(9):297–397, 2007.

[7] R. J. Cardone. *Language and Compiler Support for Mixin Programming*. PhD thesis, The University of Texas at Austin, 2002.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[9] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[10] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 302–312, 2003.

[11] J. Gil and T. Shragai. Are we ready for a safer construction environment? In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 495–519, 2009.

[12] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, Nov. 1962.

[13] J. Kusmierek. *A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming*. PhD thesis, Warsaw University, Departement of Informatics, 2010.

[14] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[15] M. Martin, M. Mezini, and S. Erdweg. Template constructors for reusable object initialization. In *GPCE'13: Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, pages 43–52, Oct. 2013.

[16] B. Meyer. Applying design by contract. *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, Oct. 1992.

[17] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[18] F. Spoto. The nullness analyser of julia. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, pages 405–424, 2010.

[19] F. Spoto and M. D. Ernst. Inference of field initialization. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 231–240, 2011.

[20] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Inf.*, 6:171–185, 1976.

[21] Y. Zibin, D. Cunningham, I. Peshansky, and V. A. Saraswat. Object initialization in X10. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 207–231, 2012.

Acknowledgements

First of all, I wish to thank professor Viviana Bono for offering me this PhD which has somehow transformed me. It has been an opportunity for academic but also personal growth. In particular, it has been an opportunity for facing some fears (while others are still to face...). I express my deep gratitude also to professors Stéphane Ducasse and Damien Cassou, for hosting me with great *accueil* within the RMoD team in Lille. I thank particularly both Viviana and Damien for believing in me, for their constant presence and care, and for their great professionalism. A special thank goes to Damien for continuing to follow my PhD even after he left the academic world for another job.

I wish to thank all the PhD students that shared with me some nice moments during this experience; all the members of both the research groups of Torino and Lille, for their kindness and availability; and Aleksy Schubert and Oscar Nierstrasz, for accepting to review my work, and for their feedbacks that helped a lot in improving the thesis.

During this PhD, some important events changed my life. My beloved Dad passed away at the age of 76. I thank him for his support in all these years of study, but even more for having conveyed me so many beautiful aspects. Aspects that now continue living within me, starting from the passion for cinema, and passing through that typical *Neapolitan humour* that so often saves me from the severity of life. This thesis is dedicated to him.

I wish to thank my Mum, with whom we support each other now. And my brother Francesco and his fiancée Eleonora, for their company and their loving support, especially during last summer.

I wish to thank my best friends Emma and Paolo. They share with me so many beautiful and ugly moments of every day, and they are always there when I get hurt. They are my safe harbour every time I am in trouble. I do not know what I would do without them.

Also my counseling ex-colleagues, and friends, have been essential during these years: Barbara, Fabrizio and Francesca. My thanks go also to them.

Special thanks go to Andrea and Paolo C., whose presence has been of great help for going through some difficult months that coincided with the writing of the first draft of this thesis. I think that life, together with difficulties, often provides people who help us to overcome troubles.

Finally, I want to thank Valérie for hosting me in her house in Loos, during my pleasant stays in Lille. She has always been very kind and friendly, as well as understanding my natural reserve. Her hospitality (along with *maroilles*) helped making me to fall in love with Lille, and making it almost a second home.